OPTIMAL SERVER SCHEDULING TO MAINTAIN CONSTANT CUSTOMER WAITING TIMES

THESIS

Thomas J. Frey
Captain, USAF

AFIT/GOR/ENS/88D-7

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY
AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT/GOR/ENS/88D-7

OPTIMAL SERVER SCHEDULING TO MAINTAIN CONSTANT CUSTOMER WAITING TIMES

THESIS

Thomas J. Frey
Captain, USAF

AFIT/GOR/ENS/88D-7

Approved for public release; distribution unlimited

AFIT/GOR/ENS/88D-7

OPTIMAL SERVER SCHEDULING TO MAINTAIN CONSTANT CUSTOMER WAITING TIMES

THESIS

Presented to the Faculty of the School of Engineering
of the Air Force Institute of Technology
Air University
In Partial Fulfillment of the
Requirements for the Degree of
Master of Science in Operations Research

Thomas J. Frey
Captain, USAF

December 1988

Preface

The purpose of this research is to develop an analytical model for scheduling check-out servers at the commissary. The goal of the model is to insure that the average customer waiting time remains constant throughout the scheduling period. Such a model can save commissary management much time and effort, and it can save money by improving the utilization of the commissary workers. The model was developed to be general enough to be used at any commissary in the Air Force, and it can also be used at many other different service organizations.

I want to extend my sincere thanks to my thesis advisor, Major Joseph Litko, for all of the help he gave me in the development of this thesis. Thanks also to my thesis reader, Dr. James Chrissis, who also provided help along the way. The biggest thanks, however, go to my wife Aneita and daughter Elizabeth, for being understanding when I did not give them the time that they deserved during the past 18 months.

Thomas J. Frey

Table of Contents

		Page
Preface		ii
List of Figures		v
Abstract		vi
I.	Introduction	1
	Background	1
	Specific Problem	2
	Research Objective	2
	Scope	3
	Plan of the Report	3
II.	Literature Review	5
	Overview	5
	Constrained Dynamic Programming	5
	The Fluid Approximation for Queues	10
	Shift Scheduling	14
	Network Programming	19
	The Out-of-Kilter Algorithm	20
	Lagrangian Relaxation	21
III.	The Checker Scheduling Algorithm	24
	Overview	24
	Phase I	24
	Phase II	30
	Lunch Breaks	33
	Lagrangian Relaxation Formulation	34
	Split-Shift Formulation	37
	Summary	40
IV.	Validation of the Model	42
	Overview	42
	Face Validity	42
	Simulation of the System	44

V. Results, Recommendations, and Conclusions 55

Results 55
Recommendations 55
Conclusions 56

Appendix A: Server Scheduler 57

Appendix B: Customer Arrival Data 86

Appendix C: SLAM Code and Output 91

Bibliography 101

Vita 103

List of Figures

Figure		Page
1.	Graphical Representation of Dynamic Programming	6
2.	Number of Customers in System as a Function of Arrivals and Departures	11
3.	Fluid Approximation to N(t)	13
4.	Segal's Network Conversion	17
5.	Modified Dynamic Program	29
6.	Equivalent Network	32
7.	Checker Requirements for 11 Jun 87	43
8.	SLAM Model	45
9.	Fluid Approximation for Entire Scheduling Period	46
10.	Fluid Approximation for 6 and 11 Jun 87	47
11.	Case 1: 6 Jun 87	48
12.	Case 2: 6 Jun 87	48
13.	Case 3: 6 Jun 87	49
14.	Case 4: 6 Jun 87	49
15.	Case 1: 11 Jun 87	50
16.	Case 2: 11 Jun 87	50
17.	Case 3: 11 Jun 87	51
18.	Case 4: 11 Jun 87	51
19.	Comparison of Approximation and Actual Customer Waiting Times	52
20.	Comparison of Approximation and Actual Customer Line Lengths	53
21.	Effect of Additional Server at Period 10	55

Abstract

The purpose of this ~~research~~ thesis was to develop an analytical model that would optimally schedule commissary checkers so that the expected customer waiting-time would remain relatively constant throughout the scheduling period. A two-phase model was developed to solve the problem. The first phase of the model used dynamic programming to find the optimal number of checkers required throughout each day to meet the desired customer waiting-time goal. Since checkers cannot be scheduled to work arbitrarily short tours of duty, a second phase was needed in the model to find the optimal number of checkers to assign to allowable shifts in order to meet the optimal requirements determined in phase one.

A simulation was developed to validate the checker scheduling model. It was found that the scheduling model produced acceptable results until the last few periods of the day. Additional servers needed to be added heuristically near the end of each day to obtain the desired customer waiting times.

Several extensions of this work are possible. First, an improved approximation for customer line lengths could be used at the end of each day. Use of such an approximation could eliminate the need for heuristic rules in scheduling servers during the last few periods of each day. Second, the scheduling algorithm that was developed did not account for checker lunch breaks. Accounting for lunch breaks complicates the problem, but two different approaches were suggested for a solution allowing for checker lunch breaks. Finally, a third phase could be added to the model that would allow assignment of actual workers to the optimal shifts determined in the second phase.

Keywords: checker scheduling algorithm, (KR)

OPTIMAL SERVER SCHEDULING TO MAINTAIN CONSTANT CUSTOMER WAITING TIMES

I. Introduction

Background

Controlling customer waiting times at service organizations such as the commissary is a difficult task. Long lines are a common cause of customer complaints. At the commissary, long lines do not usually cause customers to leave the store. However, they can result in loss of future commissary sales by causing customers to do future grocery shopping at off-base establishments. Long lines can also impair the efficiency of commissary service by creating congestion in the aisles.

Long lines are not the only problem facing commissary management. The other extreme, lines that are too short, is also a problem. From a customer's standpoint, short lines are ideal, but short lines cost the commissary extra money. To attain short lines, the commissary must employ extra check-out servers (checkers). Since each store is only allocated a set number of checker-hours each month, a store may not have the needed checker-hours available to achieve short lines. Somewhere, between long and short lines, an ideal exists. Keeping the line length and the corresponding customer waiting time at this ideal is difficult, especially in the face of limited total monthly checker-hours.

The easiest and most common way to control a queue's length is by varying the number of servers. Obviously, with more servers, shorter lines would be expected. Analytical expressions relating the number of servers to the number of customers in line are commonly available as long as certain assumptions are met. One of these assumptions is that the mean customer arrival rate remain constant. Unfortunately, the mean customer arrival rate at the commissary (and at many other service organizations) varies throughout the day, making the scheduling of servers more difficult.

Specific Problem

Current scheduling of commissary checkers requires a considerable amount of the store management's time. At the larger stores, the time spent on scheduling is estimated between eight and fourteen hours per week (Polk, 1988). Efficient allocation of servers depends mainly upon the experience of the scheduler. A reliable and automated method for scheduling the checkers would save management time and potentially save money through improved efficiency.

Research Objective

The primary objective of this research is to develop an analytical model that will optimally schedule commissary checkers so that the expected customer waiting time is constant throughout the day. Some subobjectives associated with the primary objective are:

1. Make the model sufficiently general so that it can be used at any Air Force commissary.
2. Specify a goal for the mean customer waiting time and achieve that goal throughout the month.

3. Keep the total number of checker-hours scheduled during the month below a given maximum number of allowable checker-hours for the month.

4. Validate the analytical model using simulation.

5. Observe any trends in server requirements during a month.

Scope

The monthly scheduling of checkers is a three phase problem. In Phase I, checker requirements must be determined for the scheduling period. That is, the ideal number of checkers required to achieve the mean customer waiting time goal are found in Phase I. Then, in Phase II, all possible checker shifts are enumerated, and the optimal number of checkers are assigned to each shift to meet the requirements calculated in Phase I. Finally, in Phase III, actual checkers are matched to the optimal shifts. This research effort concentrates on Phases I and II. Phase III, which is essentially an assignment problem, is left for future work.

Plan of the Report

Chapter I has introduced the problem. The background, specific problem, research objective, and scope of the research were discussed. In Chapter II, the literature pertaining to this problem is reviewed. Included in the literature review are sections discussing dynamic programming with resource constraints, fluid approximations to queues, manpower shift scheduling, integer and network programming, and application of lagrangian relaxation to integer programming. Chapter III documents the development of the checker scheduling algorithm. In Chapter IV, a

simulation is used to validate the checker scheduling algorithm. Finally, the results, conclusions, and recommendations are made in Chapter V.

II. Literature Review

Overview

There are six main areas addressed in the literature review. The first section reviews dynamic programming under constraints. The second section discusses a simple queuing approximation used to estimate customer line lengths. In the third section, various shift scheduling methods are discussed. The fourth section of the literature review shows how certain integer programming problems can be transformed into network programming problems and the fifth section outlines how these network problems can be solved. The final section of the literature review discusses the lagrangian relaxation technique for solving integer programming problems that have a special structure.

Constrained Dynamic Programming

Dynamic programming is defined by Hillier and Lieberman as "... a useful mathematical technique for making a sequence of interrelated decisions. It provides a systematic procedure for determining the combination of decisions that maximizes overall effectiveness (Hillier and Lieberman, 1986:332)." If a problem can be easily split into stages then dynamic programming should be considered as a possible solution technique. At each stage the system can be in one of a number of different states. A decision is made at the present stage. The effect of this decision is to transform the system state at the present stage into a system state at the next stage. The mechanics of this transformation are usually defined by a transition equation. A recursive function f is used so that the decisions made at each stage are optimal. The difficulty in applying

dynamic programming is in defining the recursive function f and the transition equation, which together define how to move from one stage to the next. Figure 1 shows a graphical representation of dynamic programming.

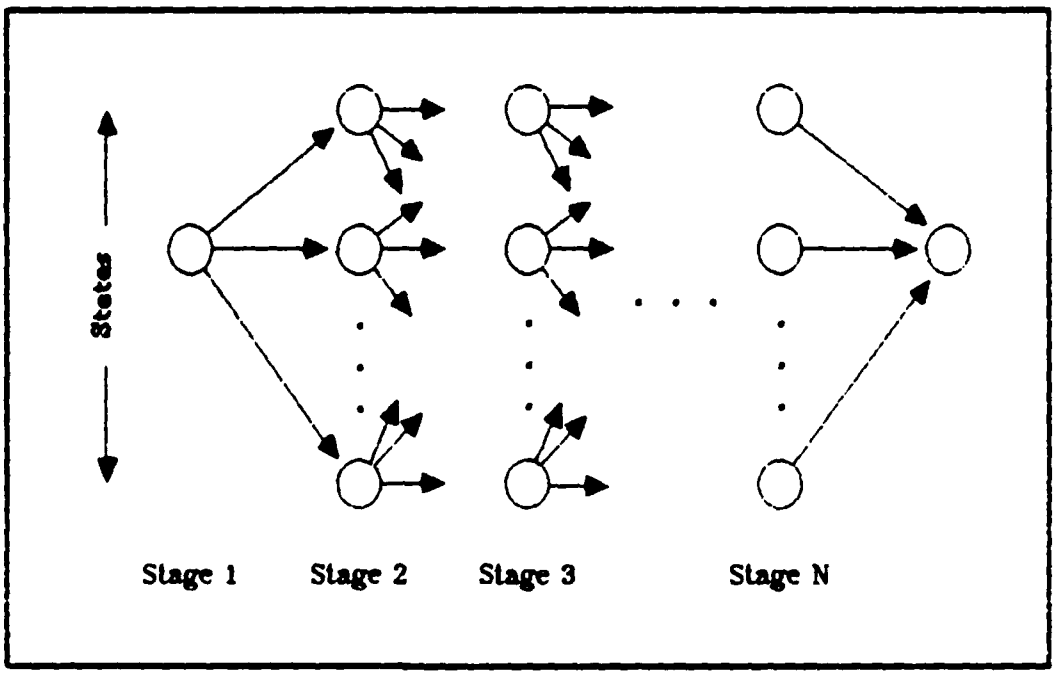


Figure 1 Graphical Representation of Dynamic Programming

One typical definition of the recursive function f at stage n is (Denardo, 1982:162):

$$f_n(i) = \begin{cases} \max_k(\min) \left[R_i^k(n) + f_{n+1}(j) \right] & \text{for } n<N \\ \max_k(\min) \left[R_i^k(N) \right] & \text{for } n=N \end{cases} \qquad (1)$$

where

n = current stage

i = system state at current stage

j = system state at next stage

N = total number of stages in the problem

k = decision made at stage n

$R_i^k(n)$ = some return function given decision k, state i, and stage n

Typically the return function is some cost function and the objective is minimization. For this case $R_i^k(n)$ would represent the cost at stage n of decision k, and $f_{n+1}(j)$ would represent the total cost of the remaining stages. Thus the current decision directly affects costs at the current stage and indirectly affects costs later by determining the next system state.

A dynamic programming problem can be solved by starting at the final stage N and working backward. The recursive function f is calculated for every possible state at stage N. To calculate f for a given state i, $R_i^k(N)$ must be calculated for every possible decision k. The recursive function $f(i)$ is then equal to the minimum value of $R_i^k(N)$ if the objective is minimization. Once f_N has been calculated for all possible states of stage N, a similar process is followed for stage $N-1$. However, now the recursion function is calculated using the first part of Eq (1):

$$f_{N-1}(i) = \min_k \left[R_i^k(N-1) + f_N(j) \right] \qquad (2)$$

One difficulty in calculating f for any stage n, where $n \neq N$, is determining the state j at the next stage ($n+1$). Recall, the state at the next stage is calculated using the transition equation. Unfortunately, there is no standard form for the transition equation. It is usually dependent upon the specific problem.

To apply dynamic programming to a queuing system, the following must be defined: the system stage (n), system state (i), the stage decision (k_n), the return function (R), the recursion equation (f), and the transition equation. Generally, the system stage (n) is some specified time period, e.g. n=1 corresponds to hour one, n=2 corresponds to hour two, and so on. The system state in a queuing system is usually the number of customers in line, and the stage decision is the number of servers to have open for the stage. Definition of the return function is not so simple. One way is to use a cost function (Magazine, 1971:178):

$$R_i^k(n) = \begin{cases} K(i)+(k_n-k_{n-1})A+k_nC & \text{if } k_n > k_{n-1} \\ K(i)+(k_{n-1}-k_n)B+k_nC & \text{if } k_n < k_{n-1} \end{cases} \qquad (3)$$

where

A = cost of opening a server

B = cost of closing a server

C = cost of operating an open server for one time period (stage)

K(i) = holding cost, i.e. cost incurred when i customers are observed in the system

A, B, and C are usually fairly easy to determine. However, it is extremely difficult to define the customer holding cost, K(i). This customer holding cost is basically an attempt to put a dollar value on the number of customers in line. The rationalization for this is that if the line is too long, business will be lost. The holding cost is a measure of the amount of the lost business. Obviously, any value used for K(i) can only be an estimate.

The computational difficulty in solving a dynamic programming problem is strongly related to the number of possible states.

Unfortunately, the addition of a resource constraint (e.g. a limit on the total hours available to be scheduled) can dramatically increase the number of possible states. The reason for this is the way that resource constraints are typically handled. Usually, an extra state variable, corresponding to the amount of resource remaining, is added (Denardo,1982:35). Recall the prior example, where the system state was given as the number of customers in line (i). When the resource constraint is added, the new system state is now a combination of the number of customers in line (i) and the amount of resource remaining (y). If the maximum number of customers in line is I and the total available increments of resource is Y, then the total possible number of states is now approximately $(I \times Y)$. This increase in the computational difficulty of a dynamic programming problem as the number of state variables is increased is known as the "curse of dimensionality" (Bellman, 1957:ix).

To avoid the curse of dimensionality in resource allocation problems, an alternative to dynamic programming is available. This algorithm is called the "maximal marginal return" procedure (Larson and Casti, 1982:350). The maximal marginal return procedure starts with none of the resource allocated. Each unit of the resource is then added so that it maximizes the immediate marginal return (or for a minimization problem, each unit of resource is added to minimize the immediate marginal return). The procedure is complete when all units of the resource are allocated. The algorithm is simple and is usually more efficient than dynamic programming. Unfortunately, a condition for its use is that the return function at each stage is independent of the return functions at all other stages. This condition is often violated in a queueing problem, where the

state of the system (number of customers in line) depends upon the actions taken at previous stages.

The Fluid Approximation for Queues (Kleinrock, 1976:56-62)

Analysis of queueing systems is complex because it involves several random variables; time between customer arrivals is a random variable, and the time required to serve a customer is a random variable. This research effort is an attempt to control queue length by varying the number of servers to the queue. To do this, the effect of the number of servers upon the queue length must be known.

Queues are generally classified by the distribution of customer interarrival times, the distribution of service times, and the number of servers for the queue. For some special distributions of customer interarrival times and service times, the exact relationship between number of servers and queue length can be derived. Probably the most well-known is the case where the distribution of the customer interarrival times is exponential and the distribution of the service times is exponential (the famous M/M/s queue). For more general cases, when the distributions are unknown or not well-behaved, approximations must be used to obtain a relationship between the number servers and the queue length. The fluid approximation is one such approximation.

In any queuing system, the number of customers is a discontinuous function of time. This is because the number of customers can only change in integral units—half a customer does not exist. The number of customers in the system at time t can be given as:

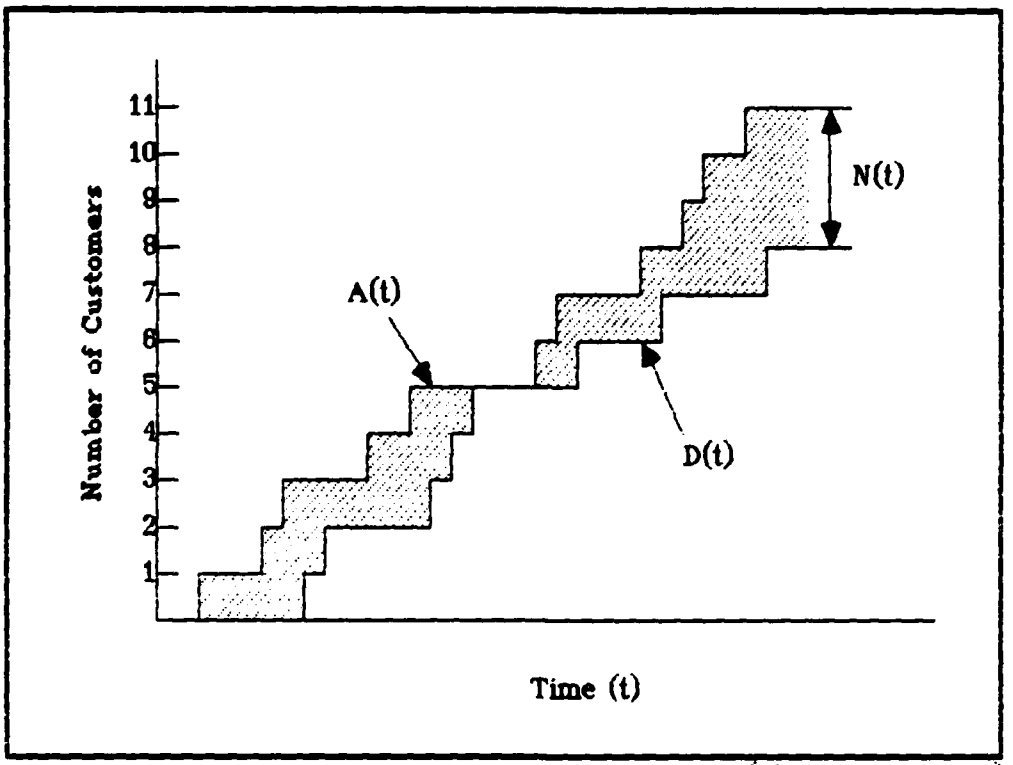
$$N(t) = A(t) - D(t) \qquad (4)$$

where

A(t) = Number of customer arrivals in (0, t)

D(t) = Number of customer departures in (0, t)

In Figure 2, the relationship between A(t), D(t), and N(t) is shown graphically.



(Kleinrock, 1976:57)

Figure 2 Number of Customers in System as a Function of Arrivals and Departures

The fluid approximation to a queue takes advantage of the fact that when a system is in a heavy traffic condition, the number of customers can be represented as a continuous function of time instead of a discontinuous function of time. This is accomplished by using the average number of

customer arrivals and departures instead of the exact values, giving the fluid approximation as:

$$N_f(t) = \overline{A(t)} - \overline{D(t)} \tag{5}$$

where

$\overline{A(t)}$ = Mean number of customer arrivals in (0, t)

$\overline{D(t)}$ = Mean number of customer departures in (0, t)

If the customer arrival rate as a function of time is given as $\lambda(t)$ and the service rate as a function of time is given as $\mu(t)$, then:

$$\overline{A(t)} = \overline{A(0)} + \int_0^t \lambda(y)dy \tag{6}$$

$$\overline{D(t)} = \overline{D(0)} + \int_0^t \mu(y)dy \tag{7}$$

The fluid approximation is shown graphically in Figure 3.

There are several limitations of the fluid approximation that should be noted. The primary assumption of the fluid approximation is that the system is in heavy traffic. If the queue empties out and servers become idle, the approximation will no longer be accurate. Problems can also arise at the other extreme—a sudden large influx of customers. The fluid approximation tends to underestimate queue length when the system reaches saturation in a very short time (the typical situation during a rush hour, where $\lambda(t) \geq \mu(t)$).

Although the queue length may be underestimated for some situations, the fluid approximation is still valid when $\lambda(t) \geq s\mu(t)$. Most attempts to schedule servers to queues use the steady-state results of

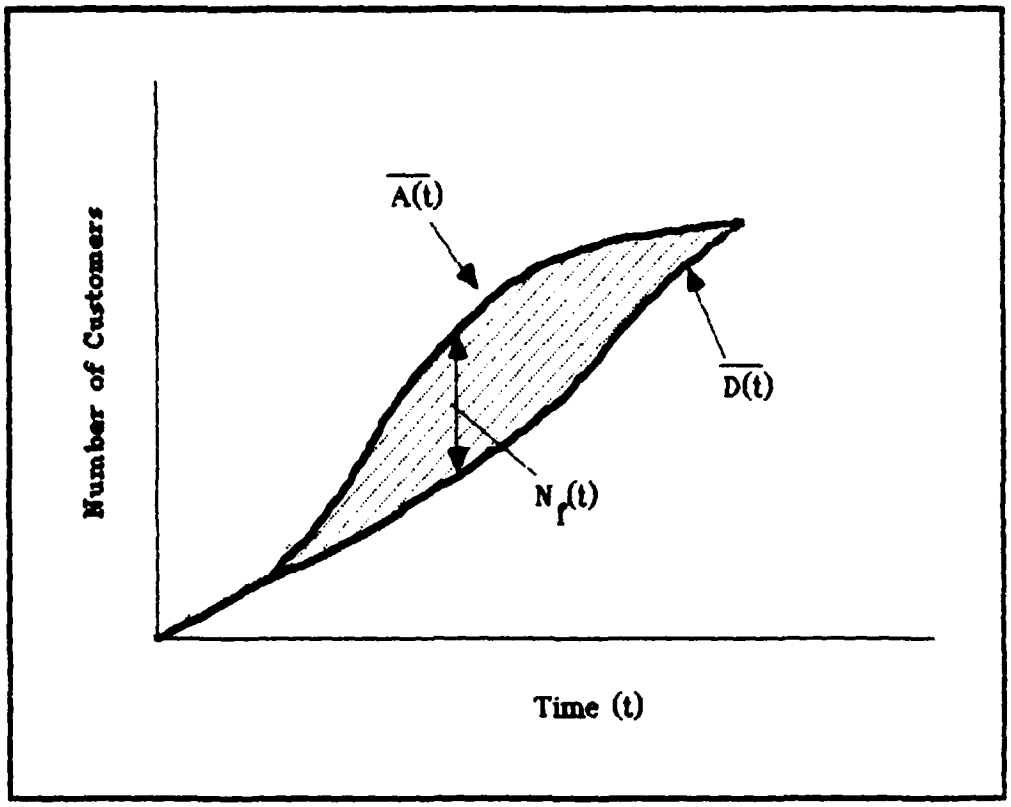


Figure 3 Fluid Approximation to N(t)

the simple M/M/s model of a queue (customer interarrival times and service times are distributed exponentially). One problem with this model is that it is only valid when:

$$\rho = \frac{\lambda}{s\mu} \leq 1 \qquad (8)$$

which means that the arrival rate cannot exceed the overall service rate. To overcome this limitation, the number of servers s must be chosen so that Eq (8) is obeyed. This is the approach used by Segal (1974) and by Kwan, Davis, and Greenwood (1988). However, there is a more fundamental problem involved with using the steady-state M/M/s model.

If the customer arrival rate and the overall service rate are changing over time, then the system never reaches steady-state (Kwan, Davis, and Greenwood, 1988:267).

The fluid approximation is useful because it is essentially a deterministic approximation to queue behavior. In other words, if one knows $A(t)$ and $D(t)$, $\overline{A(t)}$ and $\overline{D(t)}$, or $\lambda(t)$ and $\mu(t)$, then $N_f(t) \approx N(t)$ can be determined. With most other approximations, all that can be deduced is the probability distribution of $N(t)$.

Shift Scheduling

The staff scheduling problem is basically a problem of scheduling a limited resource—people—to meet a set of requirements at a minimum cost. The general formulation of this problem is (Baker, 1976:157):

$$\begin{aligned} \text{Min} \quad & \mathbf{cx} \\ \text{s.t.} \quad & \mathbf{Ax} \geq \mathbf{b} \\ & \mathbf{x} \geq \mathbf{0}, \text{ integer} \end{aligned} \tag{9}$$

where

x_j = the number of workers for shift j

c_j = the cost of assigning a worker to shift j

b_i = the number of worker required during period i

$\mathbf{A}$ = a 0-1 matrix with elements a_{ij}

$$a_{ij} = \begin{cases} 1 & \text{if shift } j \text{ works during period } i \\ 0 & \text{if shift } j \text{ does not work during period } i \end{cases}$$

Problem (9) is a general integer programming problem and can thus be difficult to solve. However, a special case of the above problem is the cyclic scheduling problem, where each column of $\mathbf{A}$ consists of consecutive ones

and zeroes. Such a case might occur when scheduling a 5-day work week where the days off must be consecutive. For this case, **A** would be given as:

$$\mathbf{A} = \begin{bmatrix} 1 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \tag{10}$$

The consecutive ones in the **A**-matrix of Eq (10), sometimes referred to as circular ones (Bartholdi, 1981:503), indicate that a worker is available continuously during all consecutive time periods. If this assumption is met, algorithms are available to solve the problem more efficiently than general integer programming (Bartholdi, 1981:503).

Segal considers the case where work periods are considered to be hours in the day instead of days in the week (Segal, 1974). For this case the **A**-matrix will be linear instead of cyclic. That is, all ones in a column of **A** will be adjacent (the top and bottom rows of **A** are not considered adjacent). In Eq (11), an example of such a matrix is shown for eight and five-hour work shifts.

To solve the problem given in (9), with **A** as in Eq (11), Segal converts the problem into a network as shown in Figure 6 (Segal, 1974:812-815). In Segal's network formulation, the nodes correspond to

$$\mathbf{A} = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \qquad (11)$$

the transition from one time period to the next. The forward arcs from i to $i+1$ correspond to the actual time periods. The backward arcs from m to l ($l < m$) represent the possible shifts. The arc parameters can be defined as follows:

Forward Arcs:

$U_{i,i+1}$ = upper capacity of arc = the maximum number of workers allowed at one time plus an estimate of the number of workers on break

$L_{i,i+1}$ = lower capacity of arc = b_i

$c_{i,i+1}$ = 0

Backward Arcs:

$U_{m,l}$ = upper capacity of arc = the maximum number of workers available to work this shift

$L_{m,l}$ = lower capacity of arc = the minimum number of workers to be assigned to work this shift

$c_{m,l}$ = the unit cost of this shift

This problem can be solved efficiently using a network flow algorithm. If the worker requirements (b_i) are all integers, then the solution is guaranteed to be integer.

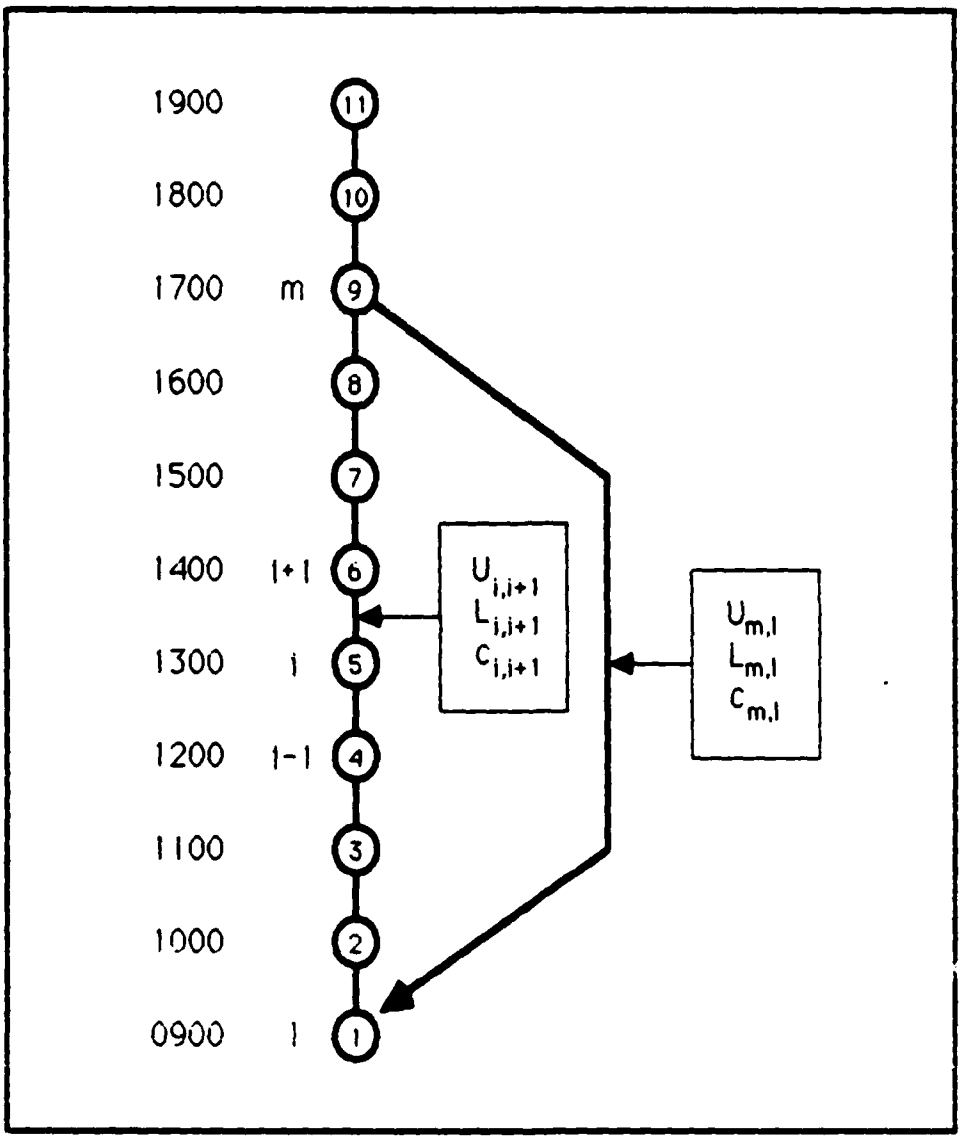


Figure 4 Segal's Network Conversion

All of the scheduling algorithms discussed thus far assume that each requirement b_i gives the minimum number of workers needed during

period i. This is why the constraints in (9) are given as inequalities. If the requirements are actually ideal numbers of workers required during each period, then the constraints in (9) should be changed to equalities. However, if this were the only change made, then the problem will often turn out to be infeasible. Therefore, an integer goal programming formulation should be used (Koelling and Bailey, 1984:302):

$$\begin{aligned} &\text{Min} \quad \bar{V} = V_1, V_2, \ldots, V_k \\ &\text{s.t.} \quad \sum_{j=1}^{n} a_{ij} x_j + d_i^- - d_i^+ = b_i \quad , \quad i = 1, \ldots, m \qquad (12) \\ &\qquad x_j \geq 0 \, , \quad j = 1, \ldots, n \end{aligned}$$

where

$\bar{V}$ = some achievement function to be specified

a_{ij} = as defined above

x_j = as defined above

b_i = as defined above

d_i^- = number of workers below requirement for time period i

d_i^+ = number of workers above requirement for time period i

Again, as long as the b_i are all integers and provided $\bar{V}$ is linear, the solution to (10) is guaranteed to be integer. Baker gives an almost identical formulation (Baker, 1976:161). The only real difference is that Baker defines the objective function specifically as:

$$\text{Minimize} \sum_{j=1}^{m} c_j x_j + \sum_{i=1}^{n} \alpha_i d_i^+ + \sum_{i=1}^{n} \beta_i d_i^- \qquad (13)$$

Network Programming (Veinott and Wagner, 1962:520)

It was no coincidence that Segal was able to convert problem (9) into a network problem. As early as 1962, Veinott and Wagner showed that any problem such as (9), where each row of the **A** matrix consists of consecutive zeroes, followed by consecutive ones, followed by consecutive zeroes, could be converted into an equivalent network problem. Suppose that **A** is $m \times n$. Each constraint except the first is replaced by itself minus the previous constraint. The first constraint is left intact. An additional constraint, equal to the last constraint times -1, is added. This method is illustrated by the following example (adapted from Veinott and Wagner, 1962:520):

$$\begin{bmatrix} 1111000000 \\ 0111111000 \\ 0011011110 \\ 0001001011 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ . \\ . \\ . \\ x_{10} \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{bmatrix} \tag{14}$$

After performing the transformation, one obtains:

$$\begin{bmatrix} 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ -1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 & -1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & -1 & 0 & 0 & -1 & 0 & -1 & 0 & 1 \\ 0 & 0 & 0 & -1 & 0 & 0 & -1 & 0 & -1 & -1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ . \\ . \\ . \\ x_{10} \end{bmatrix} = \begin{bmatrix} -b_1 \\ b_2-b_1 \\ b_3-b_2 \\ b_4-b_3 \\ -b_4 \end{bmatrix} \tag{15}$$

The equations of (15) have the required structure to be represented as a network. Each column of **A** contains a single one, a single minus one, and all remaining entries are zero, and thus **A** can be thought of as the node-arc incidence matrix of a network.

The Out-of-Kilter Algorithm (Fulkerson, 1961:18-27)

An efficient algorithm for solving problems of the form:

$$\begin{aligned} \text{Min} \quad & \mathbf{cx} \\ \text{s.t.} \quad & \mathbf{Ax} = \mathbf{b} \\ & \mathbf{0} \le \mathbf{x} \le \mathbf{u} \end{aligned} \tag{16}$$

where $\mathbf{A}$ is a node-arc incidence matrix of a network, is the out-of-kilter algorithm. The out-of-kilter algorithm is better than the simplex algorithm for problems of this form because it eliminates the need to carry the basis inverse and can thus reduce the computational burden of solving the problem.

The dual of problem (16) can be written:

$$\begin{aligned} \text{Max} \quad & \mathbf{b}\pi - \mathbf{u}\mu \\ \text{s.t.} \quad & \pi \mathbf{A} - \mu \le \mathbf{c} \\ & \mu \ge \mathbf{0} \end{aligned} \tag{17}$$

By defining:

$F(j)$ = "From" Node = the originating node of arc j

$T(j)$ = "To" Node = the destination node of arc j

$\bar{c}_j = \pi_{F(j)} - \pi_{T(j)} - c_j$

the Kuhn-Tucker conditions for optimality can be reduced to:

$$\mathbf{Ax} = \mathbf{b} \tag{18}$$

$$\text{for arc } j: \begin{cases} \bar{c}_j < 0 \text{ when } x_j = 0 \\ \bar{c}_j = 0 \text{ when } 0 \le x_j \le u_j \\ \bar{c}_j > 0 \text{ when } x_j = u_j \end{cases} \tag{19}$$

For a given ($\mathbf{x}$, $\boldsymbol{\pi}$), arc j is said to be in-kilter if (19) is satisfied. Otherwise, arc j is said to be out-of-kilter. If an arc is out-of-kilter, the kilter number is the amount of flow required to convert the arc to the in-kilter condition. For ($\mathbf{x}$, $\boldsymbol{\pi}$) to be a solution to (16), the kilter number for each arc must be zero (all arcs must be in-kilter).

The out-of-kilter algorithm consists of two phases. In the primal phase, all dual variables are held fixed, and the primal variables are changed in an attempt to reduce the sum of all kilter numbers. In the dual phase, the process is reversed. The primal variables are held fixed, and the dual variables are changed in an attempt to reduce the sum of all kilter numbers.

Lagrangian Relaxation (Fisher, 1981:1-8)

Many difficult integer programming problems can be viewed as easy problems complicated by a relatively small number of side constraints. Such a problem can be written as:

$$\begin{aligned} Z = \text{Min} \quad & \mathbf{cx} \\ \text{s.t.} \quad & \mathbf{Ax} = \mathbf{b} \\ & \mathbf{Dx} = \mathbf{e} \\ & \mathbf{x} \geq \mathbf{0}, \text{ integral} \end{aligned} \tag{20}$$

where $\mathbf{Ax} = \mathbf{b}$ are the difficult constraints. The Lagrangian relaxation is formed as follows:

$$\begin{aligned} Z_d(\mathbf{u}) = \text{Min} \quad & \mathbf{cx} + \mathbf{u}(\mathbf{Ax} - \mathbf{b}) \\ \text{s.t.} \quad & \mathbf{Dx} = \mathbf{e} \\ & \mathbf{x} \geq \mathbf{0}, \text{ integral} \end{aligned} \tag{21}$$

where $\mathbf{u} = (u_1, u_2, \ldots, u_m)$ is a vector of Lagrange multipliers. For (20) and (21), the following inequality will always hold:

$$Z_d(\mathbf{u}) \leq \mathbf{c}\mathbf{x}^* + \mathbf{u}(\mathbf{A}\mathbf{x}^* - \mathbf{b}) = Z \tag{22}$$

In general, it is not possible to find $Z_d(\mathbf{u}) = Z$. However, if $Z_d(\mathbf{u}) = Z$, then by (22), $\mathbf{x}^*$ is optimal.

Obviously, for a fixed $\mathbf{u}$, (21) is easy to solve. Ideally, $\mathbf{u}$ should be found so that $Z_d(\mathbf{u}) = Z$. The best choice of $\mathbf{u}$ is the solution to the problem:

$$Z_d = \max_{\mathbf{u}} Z_d(\mathbf{u}) \tag{23}$$

If $Z_d(\mathbf{u})$ were differentiable at all points, $\mathbf{u}$ could be found by setting the gradient of $Z_d(\mathbf{u})$ equal to 0. Unfortunately, $Z_d(\mathbf{u})$ is not differentiable at all points, so this will not work here. An adaptation of the gradient method, known as the subgradient method, has become a popular approach to selecting $\mathbf{u}$. In this method, a sequence $\{\mathbf{u}^k\}$ is generated starting at $\mathbf{u}^0 = \mathbf{0}$ and using:

$$\mathbf{u}^{k+1} = \mathbf{u}^k + t_k(\mathbf{A}\mathbf{x}^k - \mathbf{b}) \tag{24}$$

where $\mathbf{x}^k$ is an optimal solution to (21) at the previous iteration and t_k is a positive step size. A common equation for the step size is:

$$t_k = \frac{\lambda_k \left[Z^* - Z_d(\mathbf{u}^k) \right]}{\sum_{i=1}^{m} \left[\sum_{j=1}^{n} a_{ij} x_j^k - b_i \right]^2} \tag{25}$$

where λ_k is a scalar between 0 and 2. An empirical rule for λ_k is to set $\lambda_0 = 2$ and set $\lambda_k = (\lambda_{k-1}/2)$ if $Z_d(\mathbf{u})$ has failed to increase in a specified number of iterations. It should be noted that while Eq (25) has worked often in practice, there is no guarantee that it will always force convergence to the optimal solution.

Summary

This chapter presented a review of literature for six areas relevant to this research. First, dynamic programming with resource constraints was discussed. Second, the fluid approximation for obtaining an estimate of queue length was reviewed. The third area of interest in the literature review were previous efforts at shift scheduling. The fourth part of the review showed how certain integer programming problems could be converted into network problems, and the fifth part briefly discussed an efficient algorithm for solving these network problems. The sixth and final section of the literature review discussed Lagrangian relaxation, a method for removing or relaxing certain constraints that change an otherwise easily solved problem into a more difficult problem.

III. The Checker Scheduling Algorithm

Overview

The scheduling of servers at a service organization is a three phase problem. In Phase I, the server requirements for each time period must be determined so that the desired customer waiting time is obtained. Then in Phase II, the optimal number of workers to schedule to each possible shift must be found so that the server requirements determined in Phase I are met. Finally, in Phase III, actual workers are assigned to the shifts in the numbers calculated in Phase II.

The Phase I problem is solved here using dynamic programming. The output of the Phase I dynamic program becomes the input to Phase II. Because of the special structure of the constraints on the possible worker shifts, the Phase II problem can be solved using a network flow algorithm. The Phase III problem was not addressed here. One approach to the Phase III problem might be to view it as an assignment problem, where workers are assigned to the shifts calculated in Phase II.

The solution to the Phase I and Phase II problems were implemented using Turbo Pascal. The resulting code, named the Server Scheduler, is given in Appendix A.

Phase I

The purpose of Phase I is to determine the checker requirements throughout the month to obtain a mean customer waiting time of five minutes. The total checker hours must be less than a pre-set number of hours. Customer arrivals to the queue have been tabulated throughout Air Force commissaries at one-hour intervals. Because of the time-staged

nature of the arrival data, a dynamic programming approach is suggested for determining the checker requirements throughout the month. The dynamic programming approach assumes that the customer arrival rate and the service rate can be treated deterministically (the validity of this assumption is discussed in Chapter IV). It has been shown that the service time for a customer is given by (Moulder, 1987:105):

$$\text{service time} = 1.46 + 0.05\,\gamma \tag{28}$$

where γ is a random variable having a gamma distribution with parameters $\alpha = 3.2$ and $\beta = 23.1$. The mean value of γ is 73.92, so the mean service time is 5.156 minutes. The corresponding service rate μ is 0.194 customers per minute.

The natural stages in this problem are each hourly interval (n). It was not possible to formulate a return function in such a way that the return functions at each stage would be independent, and so the faster maximal marginal return method could not be used. If, however, conventional dynamic programming causes the maximum total checker hours to be exceeded, then a variant of the marginal return method will be used to deallocate checkers. As in most queueing problems, the state variable is chosen as the number of customers in line (L_n). The decision variable is the number of checkers (k_n) to have open during period n. As mentioned in Chapter II, it is usually difficult to formulate a good return function for a queueing problem. This is not the case here. Because the Air Force Commissary Service has requested that the mean customer waiting time be kept at five minutes throughout the day, the return

function can be defined as the deviation of the waiting time from five minutes, i.e.

$$R_n = |W_n - 5| \tag{27}$$

where W_n is the mean customer waiting time during period n. The mean customer waiting time can be defined as:

$$W_n = \frac{L_n + L_{n+1}}{2\mu k_n} \tag{28}$$

where

L_n = number of customers in line at stage n

L_{n+1} = number of customers in line at stage n+1

μ = mean service rate

k_n = number of checkers at stage n

In Eq (27), $[(L_n + L_{n+1}) / 2]$ gives the average number of customers in the one-hour interval while μk_n gives the overall service rate of all checkers combined. The number of customers in line at stage n+1 is related to the number of customers in line at stage n by:

$$L_{n+1} = \begin{cases} L_n + A_n - 60\mu k_n & \text{if n} \neq \text{final hour of day} \\ 0 & \text{if n = final hour of day} \end{cases} \tag{29}$$

where all variables are as defined above, and A_n is the number of customer arrivals during the one-hour interval (which has been tabulated). Eq (29) is a fluid approximation to the behavior of the queue, with arrivals $A(t) = A_n$ and departures $D(t) = 60\mu k_n$. The case of $L_{n+1} = 0$ is used to force the queue to empty at the end of each day and start empty for the following day. For this to occur, the departures in the last period of the

day must exceed the total number of leftover customers plus the number of arrivals in the last period, i.e. $60\mu k_{n+1} \geq L_n + A_{n+1}$. Thus the normal flow of a dynamic program as shown in Figure 1 is modified as shown in Figure 5.

The only other formula needed to complete the dynamic programming formulation is the backward recursion formula. It is as given in Eq (1), where the objective is minimization:

$$f_n = \begin{cases} \min\limits_{k_n} \left[R_n + f_{n+1} \right] & \text{for } n < N \\ \min\limits_{k_N} \left[R_N \right] & \text{for } n = N \end{cases} \tag{30}$$

The dynamic programming formulation of the Phase I problem can be summarized as follows:

Stage Variable

n = each one-hour interval

State Variable

L_n = number of customers in line

Decision Variable

k_n = number of checkers open

Transition Equation

$$L_{n+1} = \begin{cases} L_n + A_n - 60\mu k_n & \text{if } n \neq \text{final hour of day} \\ 0 & \text{if } n = \text{final hour of day} \\ & \text{and } 60\mu k_n \geq L_{n+1} + A_n \end{cases}$$

Return Function

$$R_n = |W_n - 5| = \left| \frac{L_n + L_{n+1}}{2\mu k_n} - 5 \right| = \left| \frac{2L_n + A_n - 60\mu k_n}{2\mu k_n} - 5 \right|$$

Recursion Formula

$$f_n = \begin{cases} \min\limits_{k_n} \left[R_n + f_{n+1} \right] & \text{for } n<N \\ \min\limits_{k_N} \left[R_N \right] & \text{for } n=N \end{cases}$$

The Phase I dynamic programming technique is implemented in the Server Scheduler in the procedure "Phase1." This procedure in turn makes calls to procedures to calculate L_{n+1} ("calc_num_cust"), W_n ("waiting_time"), and R_n ("Return").

The output of Phase I is a the vector k containing the checker requirements needed throughout the month to achieve a five minute waiting time. If the total checker hours exceed the maximum total hours, i.e.

$$\sum_{n=1}^{N} k_n > \text{Max Total Hours} \qquad (31)$$

then checkers must be removed until the total checker hours are equal to the maximum total hours. A variation of the maximal marginal return method is used to determine the stage from which to remove a checker. Checkers are removed, one at a time, from the stage that produces the least gain in the overall sum of the return functions. In the Server Scheduler, checker removal via the marginal return method is accomplished in the procedure "Deallocate."

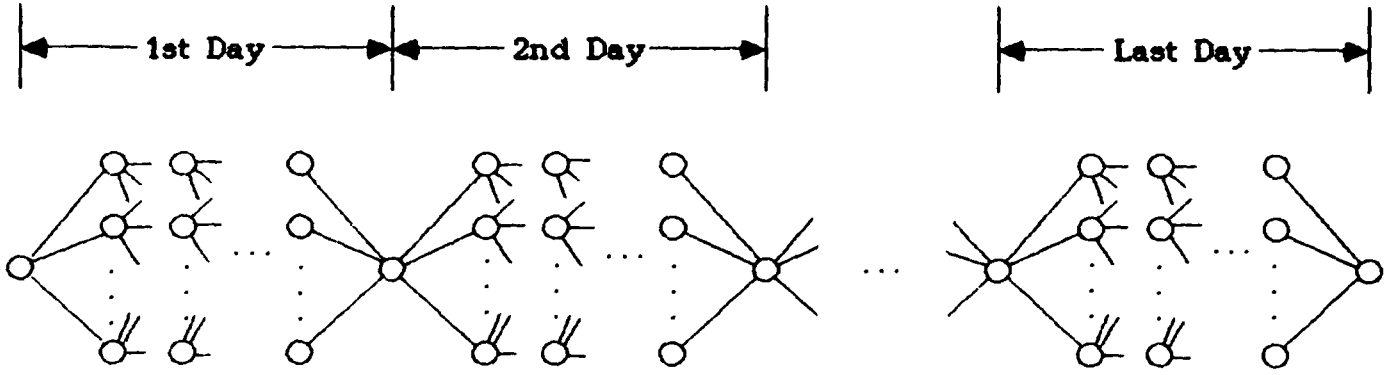


Figure 5 Modified Dynamic Program

Phase II

Once the checker requirements are calculated in Phase I, the optimal shift schedules can be determined. Because of the number of possible shifts, it is best to solve Phase II for each day separately and then combine the daily shift schedules into an overall monthly schedule. It is assumed that possible shift lengths are 4, 5, 6, 7, or 8 hours long.

The Phase II problem is formulated as in equations (12) and (13):

$$\begin{aligned} \text{Min} \quad & \mathbf{1d^+ + 1d^-} \\ \text{s.t.} \quad & \mathbf{Ax + d^- - d^+ = b} \\ & \mathbf{x \geq 0}, \text{ integer} \\ & \mathbf{d^- \geq 0}, \text{ integer} \\ & \mathbf{d^+ \geq 0}, \text{ integer} \end{aligned} \tag{32}$$

where all variables are as defined in Chapter II. A sample constraint matrix for problem (32) is given in Eq (33). For simplicity, the sample constraint matrix shows only shift lengths of 4 and 8 hours.

$$\begin{pmatrix} 1\,0\,0\,0\,1\,0\,0\,0\,0\,0\,0\,0 & 1\text{-}1\,0\,0\,0\,0\,0\,0\,0\,0\,0 & 0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0 \\ 1\,1\,0\,0\,1\,1\,0\,0\,0\,0\,0\,0 & 0\,0\,1\text{-}1\,0\,0\,0\,0\,0\,0\,0 & 0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0 \\ 1\,1\,1\,0\,1\,1\,1\,0\,0\,0\,0\,0 & 0\,0\,0\,0\,1\text{-}1\,0\,0\,0\,0\,0 & 0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0 \\ 1\,1\,1\,1\,1\,1\,1\,1\,0\,0\,0\,0 & 0\,0\,0\,0\,0\,0\,1\text{-}1\,0\,0\,0 & 0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0 \\ 1\,1\,1\,1\,0\,1\,1\,1\,1\,0\,0\,0 & 0\,0\,0\,0\,0\,0\,0\,0\,1\text{-}1\,0 & 0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0 \\ 1\,1\,1\,1\,0\,0\,1\,1\,1\,1\,0\,0 & 0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,1 & \text{-}1\,0\,0\,0\,0\,0\,0\,0\,0\,0 \\ 1\,1\,1\,1\,0\,0\,0\,1\,1\,1\,1\,0 & 0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0 & 0\,1\text{-}1\,0\,0\,0\,0\,0\,0\,0 \\ 1\,1\,1\,1\,0\,0\,0\,0\,1\,1\,1\,1 & 0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0 & 0\,0\,0\,1\text{-}1\,0\,0\,0\,0\,0 \\ 0\,1\,1\,1\,0\,0\,0\,0\,0\,1\,1\,1 & 0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0 & 0\,0\,0\,0\,1\text{-}1\,0\,0\,0\,0 \\ 0\,0\,1\,1\,0\,0\,0\,0\,0\,0\,1\,1 & 0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0 & 0\,0\,0\,0\,0\,0\,1\text{-}1\,0\,0 \\ 0\,0\,0\,1\,0\,0\,0\,0\,0\,0\,0\,1 & 0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0 & 0\,0\,0\,0\,0\,0\,0\,1\text{-}1 \end{pmatrix} \begin{pmatrix} x \\ d- \\ d+ \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \\ b_8 \\ b_9 \\ b_{10} \\ b_{11} \end{pmatrix} \tag{33}$$

Using the method of Veinott and Wagner (1962) as outlined in Chapter II, the constraints are transformed into a network node–arc incidence matrix.

The constraints are illustrated in Eq (34), and the corresponding network is shown in Figure 6.

$$
\begin{pmatrix}
1&0&0&0&1&0&0&0&0&0&0&0&1&-1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&1&0&0&0&1&0&0&0&0&0&0&-1&1&1&-1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&1&0&0&0&1&0&0&0&0&0&0&0&-1&1&1&-1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&1&0&0&0&1&0&0&0&0&0&0&0&0&-1&1&1&-1&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&-1&0&0&0&1&0&0&0&0&0&0&0&0&0&-1&1&1&-1&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&-1&0&0&0&1&0&0&0&0&0&0&0&0&0&0&-1&1&1&-1&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&-1&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&-1&1&1&-1&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&-1&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&-1&1&1&-1&0&0&0&0&0&0\\
-1&0&0&0&0&0&0&0&-1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&-1&1&1&-1&0&0&0&0\\
0&-1&0&0&0&0&0&0&0&-1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&-1&1&1&-1&0&0\\
0&0&-1&0&0&0&0&0&0&0&-1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&-1&1&1&-1\\
0&0&0&-1&0&0&0&0&0&0&0&-1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&-1&1
\end{pmatrix}
\begin{pmatrix} \mathbf{x} \\ d- \\ d+ \end{pmatrix}
=
\begin{pmatrix} b_1 \\ b_2-b_1 \\ b_3-b_2 \\ b_4-b_3 \\ b_5-b_4 \\ b_6-b_5 \\ b_7-b_6 \\ b_8-b_7 \\ b_9-b_8 \\ b_{10}-b_9 \\ b_{11}-b_{10} \\ -b_{11} \end{pmatrix}
\qquad (34)
$$

Since the network problem given in Eq (34) and Figure 6 is equivalent to the problem given in Eq (33), the solution to the network problem is the solution to the original integer goal programming problem. But the solution to the network problem can be found much more quickly than the solution to the integer programming problem. More importantly, since the elements of **b** are all integer, the solution to (32) is guaranteed to be integral if the constraints are as in Eqs (33) and (34). In the Server Scheduler, the out-of-kilter algorithm is used to solve the equivalent network programs. The procedure used is called "MinCostFlow," and is based on the implementation of the out-of-kilter algorithm given in Kennington and Helgason (1980:78-88).

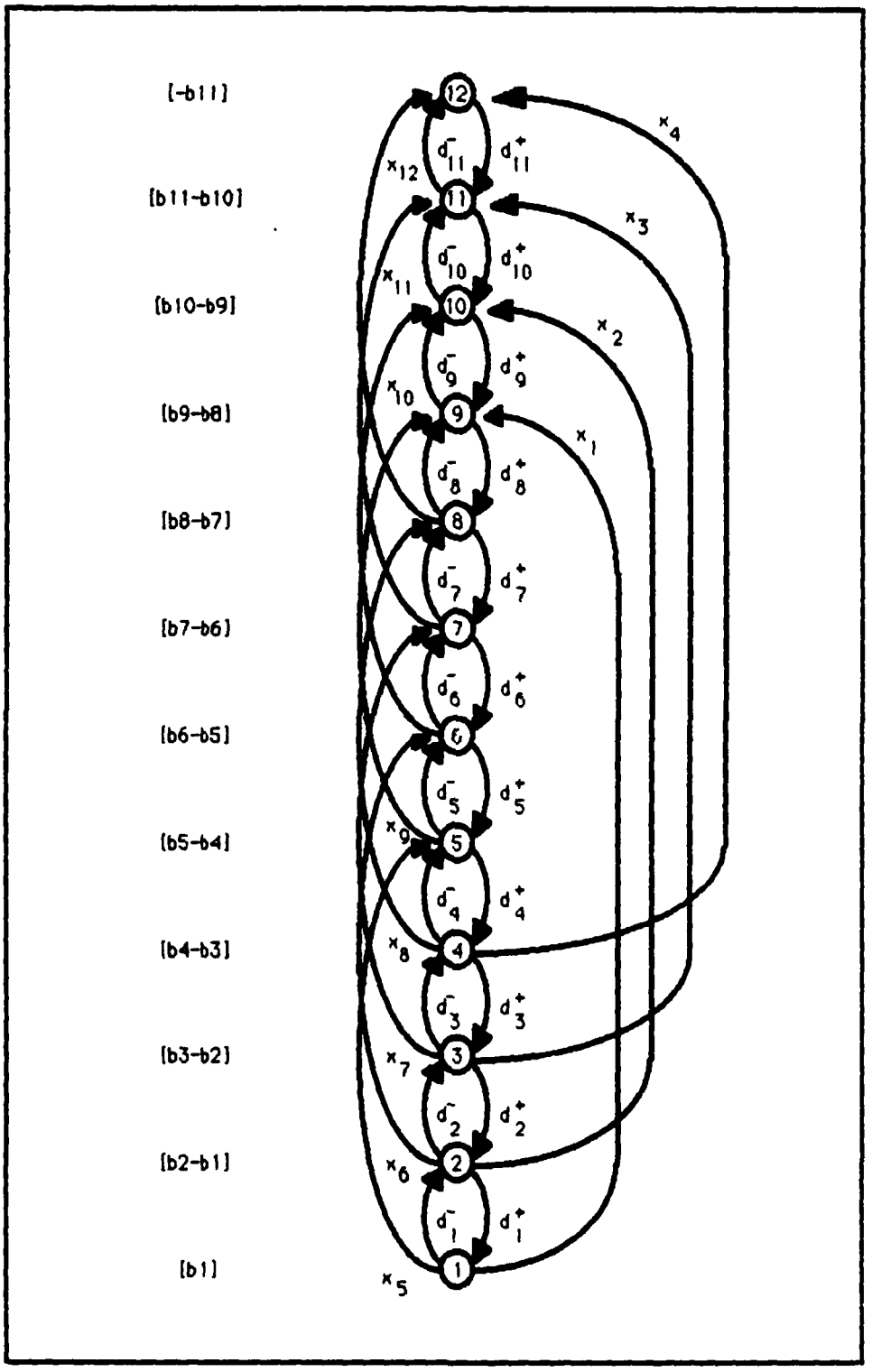


Figure 6 Equivalent Network

Lunch Breaks

Because a special network flow algorithm can be used to solve it, the Phase II procedure outlined above is very efficient. However, this formulation has not made provision for workers to take lunch breaks. Assuming that a worker on any shift longer than six hours is entitled to a one-hour lunch break, the sample constraint matrix shown in Eq (33) must be modified as shown in Eq (35):

$$
\begin{pmatrix}
1&0&0&1&0&0&0&0&0&0&0&1&-1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
1&1&0&1&1&0&0&0&0&0&0&0&0&1&-1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
1&1&1&1&1&1&0&0&0&0&0&0&0&0&0&1&-1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
1&1&1&1&1&1&1&0&0&0&0&0&0&0&0&0&0&1&-1&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&1&1&0&1&1&1&1&0&0&0&0&0&0&0&0&0&0&0&1&-1&0&0&0&0&0&0&0&0&0&0&0&0\\
1&0&1&0&0&1&1&1&1&0&0&0&0&0&0&0&0&0&0&0&0&1&-1&0&0&0&0&0&0&0&0&0&0\\
1&1&0&0&0&0&1&1&1&1&0&0&0&0&0&0&0&0&0&0&0&0&0&1&-1&0&0&0&0&0&0&0&0\\
1&1&1&0&0&0&0&1&1&1&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&-1&0&0&0&0&0&0\\
1&1&1&0&0&0&0&0&1&1&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&-1&0&0&0&0\\
0&1&1&0&0&0&0&0&0&1&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&-1&0&0\\
0&0&1&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&-1
\end{pmatrix}
\begin{pmatrix} x \\ d- \\ d+ \end{pmatrix}
=
\begin{pmatrix} b1 \\ b2 \\ b3 \\ b4 \\ b5 \\ b6 \\ b7 \\ b8 \\ b9 \\ b10 \\ b11 \end{pmatrix}
\qquad (35)
$$

This new constraint matrix is no longer amenable to the network transformation, and the solution to the problem is no longer guaranteed to be integral. Therefore, to solve this problem one must resort to more difficult general integer programming techniques. For a typical day, when five-, six-, and seven-hour shifts are added, the problem given in (35) would have 50 to 60 integer variables that range from 0 to 30. Each individual variable would need five 0–1 variables (2^5=32). This gives a total of 250 to 300 variables in a 0–1 integer programming formulation (too large to solve on a microcomputer).

Several alternate formulations are available that can help avoid the problems associated the formulation of (35). The first alternate formulation adds additional constraints for every hour corresponding to a

shift lunch hour. These constraints are then dualized to form a Lagrangian relaxation to the original problem. The Lagrangian relaxation retains the consecutive ones in the constraint matrix, thus guaranteeing an all-integer solution. The second alternate formulation splits each eight-hour shift into two four-hour shifts, each seven-hour shift into a four and a three hour shift, and each six-hour shift into two three hour shifts. This formulation also retains the consecutive ones in the constraint matrix, and the solution is again guaranteed to be all-integer.

Lagrangian Relaxation Formulation. In this formulation, as in the formulation of (35), the length of a shift with a lunch hour is increased by one hour. An eight-hour shift is changed into a nine-hour shift, a seven-hour shift is increased into an eight-hour shift, and a six-hour shift becomes a seven-hour shift. Unlike the formulation of (35), the added hour in a shift is not a zero in the constraint matrix. The new formulation is given in (36):

$$\text{Min} \quad \mathbf{1}d^- + \mathbf{1}d^+ \tag{36}$$

$$\begin{pmatrix}
1&0&0&1&0&0&0&0&0&0&0&1&-1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
1&1&0&1&1&0&0&0&0&0&0&0&0&1&-1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
1&1&1&1&1&1&0&0&0&0&0&0&0&0&0&1&-1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
1&1&1&1&1&1&1&0&0&0&0&0&0&0&0&0&1&-1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
1&1&1&0&1&1&1&1&0&0&0&0&0&0&0&0&0&1&-1&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
1&1&1&0&0&1&1&1&1&0&0&0&0&0&0&0&0&0&1&-1&0&0&0&0&0&0&0&0&0&0&0&0&0\\
1&1&1&0&0&0&1&1&1&1&0&0&0&0&0&0&0&0&0&1&-1&0&0&0&0&0&0&0&0&0&0&0&0\\
1&1&1&0&0&0&0&1&1&1&1&0&0&0&0&0&0&0&0&0&1&-1&0&0&0&0&0&0&0&0&0&0&0\\
1&1&1&0&0&0&0&0&1&1&1&1&0&0&0&0&0&0&0&0&0&1&-1&0&0&0&0&0&0&0&0&0&0\\
1&1&1&0&0&0&0&0&0&1&1&1&1&0&0&0&0&0&0&0&0&0&1&-1&0&0&0&0&0&0&0&0&0\\
0&1&1&0&0&0&0&0&0&1&1&1&1&0&0&0&0&0&0&0&0&0&0&1&-1&0&0&0&0&0&0&0&0\\
0&0&1&0&0&0&0&0&0&0&1&1&1&1&0&0&0&0&0&0&0&0&0&0&1&-1&0&0&0&0&0&0&0\\
0&0&1&0&0&0&0&0&0&0&0&1&1&1&1&0&0&0&0&0&0&0&0&0&0&1&-1&0&0&0&0&0&0\\
0&0&1&0&0&0&0&0&0&0&0&0&1&1&1&1&0&0&0&0&0&0&0&0&0&0&1&-1&0&0&0&0&0\\
0&0&1&0&0&0&0&0&0&0&0&0&0&1&1&1&1&0&0&0&0&0&0&0&0&0&0&1&-1&0&0&0&0\\
0&0&1&0&0&0&0&0&0&0&0&0&0&0&1&1&1&1&0&0&0&0&0&0&0&0&0&0&1&-1&0&0&0\\
0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&1&1&1&1&0&0&0&0&0&0&0&0&0&0&1&-1&0&0\\
0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&1&1&1&1&0&0&0&0&0&0&0&0&0&0&1&-1&0\\
0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&1&1&1&0&0&0&0&0&0&0&0&0&0&1&-1
\end{pmatrix}
\begin{pmatrix} x \\ d^- \\ d^+ \end{pmatrix}
=
\begin{pmatrix} b1 \\ b2 \\ b3 \\ b4 \\ b5 \\ b6 \\ b7 \\ b8 \\ b9 \\ b10 \\ b11 \end{pmatrix}$$

The constraint matrix of (36) is not complete. If the only the constraints of (36) were used, the number of checkers during the fifth,

sixth, and seventh hours would be overestimated by x_1, x_2, and x_3 respectively. The reason for this is that the checkers assigned to these shifts are actually out-to-lunch during these hours. This is best illustrated by example. During the fifth hour, the constraints of (36) indicate that the number of checkers on duty is:

$$x_1 + x_2 + x_3 + x_5 + x_6 + x_7 + x_8 \tag{37}$$

Since checkers from the first shift are out to lunch, the actual number of checkers on duty is:

$$x_2 + x_3 + x_5 + x_6 + x_7 + x_8 \tag{38}$$

In other words, requirement b_5 will be undershot by x_1. To off-set the overestimation of checkers, extra constraints are added to force extra checkers to be scheduled during lunch hours. For the example given above, the extra constraint is of the form:

$$d_5^+ = x_1 \quad \text{or} \quad d_5^+ - x_1 = 0 \tag{39}$$

The effect of the added constraint is to force x_1 extra checkers to be scheduled during the fifth hour. These extra checkers exactly offset the shortage created by the checkers that are out to lunch. The complete formulation for the sample problem is given in (40):

$$\text{Min} \sum_{i=1}^{11} d_i^- + d_i^+ \tag{40}$$

$$\begin{pmatrix}
1&0&0&1&0&0&0&0&0&0&0&1&-1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
1&1&0&1&1&0&0&0&0&0&0&0&0&1&-1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
1&1&1&1&1&1&0&0&0&0&0&0&0&0&0&1&-1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
1&1&1&1&1&1&1&0&0&0&0&0&0&0&0&0&0&1&-1&0&0&0&0&0&0&0&0&0&0&0&0&0\\
1&1&1&0&1&1&1&1&0&0&0&0&0&0&0&0&0&0&1&-1&0&0&0&0&0&0&0&0&0&0&0&0\\
1&1&1&0&0&1&1&1&1&0&0&0&0&0&0&0&0&0&0&1&-1&0&0&0&0&0&0&0&0&0&0&0\\
1&1&1&0&0&0&1&1&1&1&0&0&0&0&0&0&0&0&0&0&1&-1&0&0&0&0&0&0&0&0&0&0\\
1&1&1&0&0&0&0&1&1&1&1&0&0&0&0&0&0&0&0&0&0&1&-1&0&0&0&0&0&0&0&0&0\\
1&1&1&0&0&0&0&0&1&1&1&1&0&0&0&0&0&0&0&0&0&0&1&-1&0&0&0&0&0&0&0&0\\
1&1&1&0&0&0&0&0&0&1&1&1&1&0&0&0&0&0&0&0&0&0&0&1&-1&0&0&0&0&0&0&0\\
0&1&1&0&0&0&0&0&0&1&1&1&1&0&0&0&0&0&0&0&0&0&0&0&1&-1&0&0&0&0&0&0\\
0&0&1&0&0&0&0&0&0&1&1&1&1&0&0&0&0&0&0&0&0&0&0&0&0&1&-1&0&0&0&0&0
\end{pmatrix}
\begin{pmatrix} x \\ d^- \\ d^+ \end{pmatrix}
\cdot
\begin{pmatrix} b1 \\ b2 \\ b3 \\ b4 \\ b5 \\ b6 \\ b7 \\ b8 \\ b9 \\ b10 \\ b11 \end{pmatrix}$$

$$d_5^+ - x_1 = 0$$

$$d_6^+ - x_2 = 0$$

$$d_7^+ - x_3 = 0$$

The extra lunch-hour constraints in (40) destroy the consecutive ones property that allowed the problem to be converted into a network (and insured an integer solution). To regain the consecutive ones property, the lunch-hour constraints are dualized and a Lagrangian relaxation is formed:

$$\text{Min} \left[\sum_{i=1}^{11} (d_i^- + d_i^+) \right] + u_1(d_5^+ - x_1) + u_2(d_6^+ - x_2) + u_3(d_7^+ - x_3) \qquad (41)$$

$$\begin{pmatrix}
1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 1 & 1 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0
\end{pmatrix}
\begin{pmatrix} x \\ d- \\ d+ \end{pmatrix} =
\begin{pmatrix} b1 \\ b2 \\ b3 \\ b4 \\ b5 \\ b6 \\ b7 \\ b8 \\ b9 \\ b10 \\ b11 \end{pmatrix}$$

Theoretically, the problem formulated in (41) should be solvable by the method outlined in Chapter 2. First, the multipliers are set at $\mathbf{u}^0 = (0,0,0)$, and the resulting problem can be solved using the network flow algorithms already outlined. Then the multipliers are updated using Eqs (24) and (25) and the process is repeated until a solution is found that satisfies the extra lunch break constraints ($t_k = 0$ in Eq (25)). Unfortunately, implementation of the Lagrangian relaxation method for this problem proved to be difficult. Apparently, this is one application for which Eq (25) failed to produce convergence to the optimal solution. Lagrangian relaxation is still a promising method for solving the shift scheduling problem with additional constraints (including but not limited to lunch break constraints). However, a different policy must be found to update the multipliers, since Eq (25) has proven to be ineffective for this application.

Split-Shift Formulation. Another promising way to account for lunch breaks is to break each shift requiring a lunch break into two shifts. For example, an eight-hour shift extending from 0900 to 1800 with a lunch

break at 1300 would become two four-hour shifts, the first extending from 0900 to 1300 and the second from 1400 to 1800. Similarly, seven-hour shifts are split into a four and a three-hour shift, and six-hour shifts become two three-hour shifts. Using this formulation, the final constraint matrix consists of five, four, and three-hour shifts:

$$\text{Min} \left[\sum_{i=1}^{11} (d_i^- + d_i^+) \right] \tag{42}$$

$$\begin{pmatrix} 1000000100000001000000001-1000000000000000000000 \\ 1100000110000001100000000001-1000000000000000000000 \\ 1110000111000001110000000001-1000000000000000000000 \\ 1111000011110000111000000000001-1000000000000000000 \\ 1111100011111000011100000000000001-1000000000000000 \\ 0111110001111100001111000000000000001-1000000000000 \\ 0011111000111110000111100000000000000001-1000000000 \\ 0001111100001111100001111000000000000000001-1000000 \\ 0000111100001111000001111000000000000000001-10000 \\ 0000011100000111000000111100000000000000000001-100 \\ 0000001100000011000000011100000000000000000000001-1 \end{pmatrix} \begin{pmatrix} x \\ d- \\ d+ \end{pmatrix} = \begin{pmatrix} b1 \\ b2 \\ b3 \\ b4 \\ b5 \\ b6 \\ b7 \\ b8 \\ b9 \\ b10 \\ b11 \end{pmatrix}$$

The split-shift formulation is not without complications. First, solution of the problem given in (42) gives optimal numbers of five-, four-, and three-hours shifts. Some way must be found to convert these shifts back into eight-, seven-, six-, five-, and four-hour shifts. Since this conversion could be done by hand if need be, this is not a serious limitation. But a serious limitation does exist. If three-hour shifts are not allowable, one must ensure that all three-hour shifts in the optimal solution can be converted into six, seven, and eight-hour shifts. There are at least two possible ways to ensure this. The first way maintains the network structure but does not guarantee that a solution could be found, while the

second guarantees a solution but requires a Lagrangian relaxation problem to be solved.

The first method for ensuring the conversion of three-hour shifts is similar to branch-and-bound methods for solving integer programs. The problem of (42) is first solved as given using the network algorithm (as before). If all three-hour shifts can be converted into six-, seven-, and eight-hour shifts, the problem is solved. If, however, there are M_3 extra three-hour shifts that can not be converted from shift x_m, then an upper bound of $x_3 - M_3$ is placed on x_3, and the problem is solved again. The process is continued until a feasible solution is obtained.

The second method for ensuring that the three-hour shifts can be converted consists of adding extra constraints. For each three-hour shift, all other shifts that could combine with the three-hour shift to form a six-, seven-, or eight-hour shift are enumerated. The sum of the checkers assigned to these other shifts must exceed the number of checkers assigned to work the three-hour shifts. For the problem of (42), this means that the following constraints must be added:

$$x_{16} \leq x_{12} + x_{20} \tag{43}$$

$$x_{17} \leq x_{13} + x_{21} \tag{44}$$

$$x_{18} \leq x_{14} + x_{22} \tag{45}$$

$$x_{19} \leq x_{15} + x_{23} \tag{46}$$

$$x_{20} \leq x_{16} + x_{24} \tag{47}$$

$$x_{21} \leq x_8 + x_{17} \tag{48}$$

$$x_{22} \leq x_9 + x_{18} \tag{49}$$

$$x_{23} \leq x_{10} + x_{19} \tag{50}$$

$$x_{24} \leq x_{11} + x_{20} \qquad (51)$$

These extra constraints again destroy the network structure of the problem. To regain that structure, one can again apply a Lagrangian relaxation, with the same complications experienced in the original Lagrangian relaxation formulation. Of course, if three hour shifts are allowable, then the formulation given in (42) can be used, and the optimal solution would be obtained.

Summary

The Server Scheduler Program implements a two-phase algorithm for scheduling servers at a service organization (specifically, checkers at U.S. Air Force commissaries). In Phase I of the algorithm, dynamic programming is used to find the number of servers required during each scheduling period to obtain a target customer waiting time. A fluid approximation to queue length is used to calculate the average customer waiting time during each period. Then, in Phase II of the algorithm, the optimal number of servers to schedule to each possible shift is found so that the requirements determined in Phase I are met. Integer programming was used to implement Phase II of the scheduling algorithm. Because of the special structure of the constraints in the Phase II integer program, network techniques could be used to solve Phase II efficiently.

Scheduling of checker lunch breaks is a difficult extension to the Phase II shift scheduling problem. A brute force approach, simply adding zeroes to the corresponding row of each shift, destroys the structure of the problem. With this approach, applying linear programming to the problem is not guaranteed to produce an integer solution, so more difficult and

time–consuming integer programming methods must be used. Use of Lagrangian relaxation regains the special structure that allows efficient network techniques to be used iteratively to solve the problem. However, commonly used multiplier update formulas do not work for this problem, so more research is needed to determine if a Lagrangian relaxation technique can be successfully applied here. Finally, a split–shift formulation was explored. In this formulation, each shift with a lunch break was split into two shifts. Again, the special network structure was regained that would allow this formulation to be solved efficiently. If three–hour shifts are allowable, this formulation works. However, if three–hour shifts are not allowed, then Lagrangian relaxation techniques must be used to solve this formulation.

IV. Validation of the Model

Overview

This chapter is concerned with the validity of the Phase I part of the Server Scheduler model, i.e. the determination of the optimal checker requirements. Two different types of model validation are discussed here. First, face validity of the model is briefly explored. That is, does the model and its output seem to make sense? After face validity is checked, a simulation is used to see if the model achieves its goals—specifically the achievement of the desired mean customer waiting time. The simulation can also be used to investigate the general behavior of the system under some typical conditions.

The data used to test the model was tabulated for three weeks in May and June of 1987 at the Lackland AFB Commissary. It consisted of hourly counts of the number of customers arriving to the queue and the number of customers currently in the queue. This data is given in Appendix B.

Face Validity

Using the three weeks of data given in Appendix B for customer arrivals, Phase I of the Server Scheduler was run to determine the optimal checker requirements. In Figure 16, the checker requirements k_n and the arrivals A_n are plotted against n for a typical day. As would be expected, when the number of customer arrivals increases, more checkers are required. Also, when the number of customer arrivals in a period is very large, extra checkers are scheduled in the preceding periods in an

attempt to prepare for the surge. So from a simple face validity standpoint, the output of the Server Scheduler is consistent and sensible.

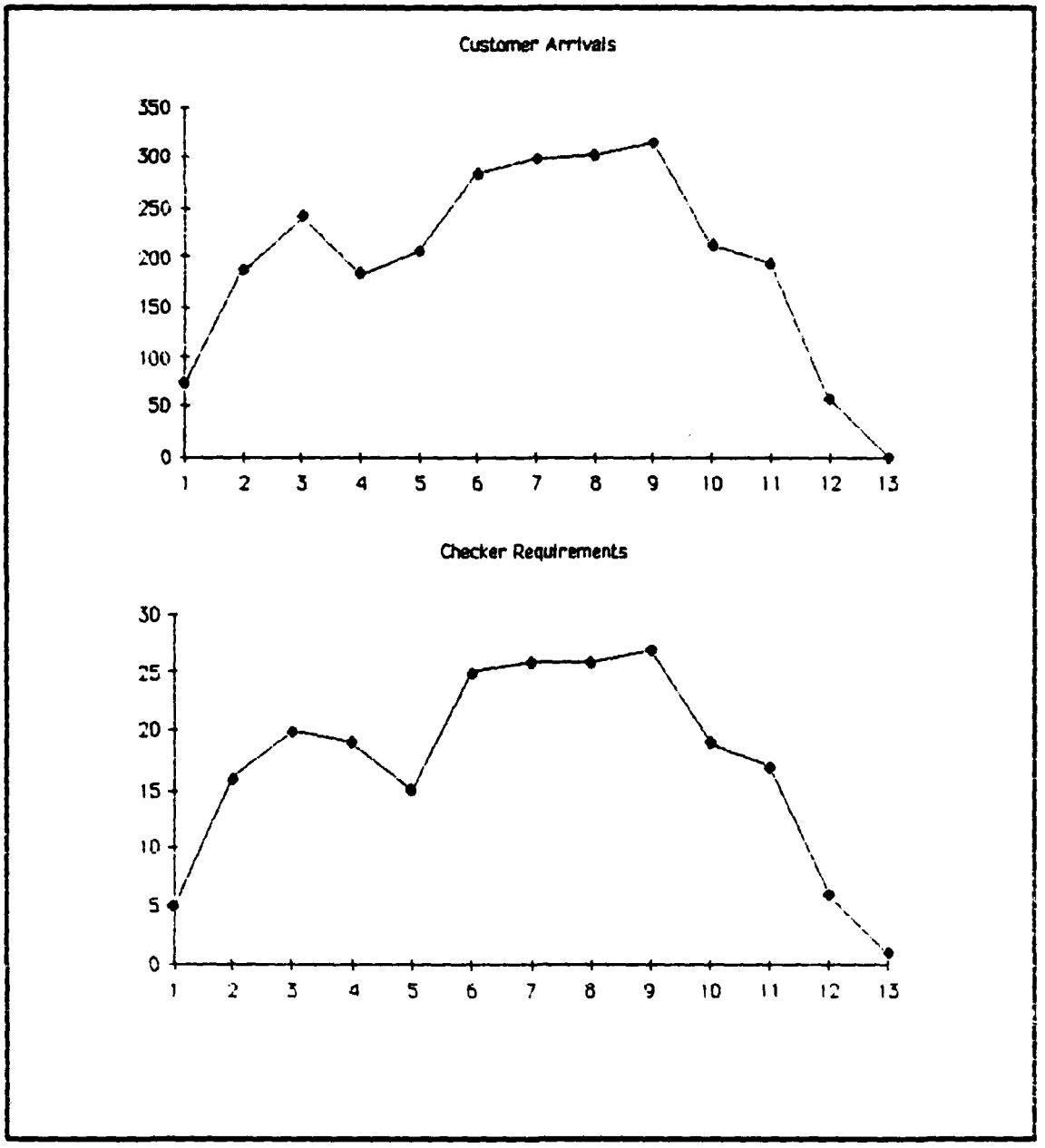


Figure 7 Checker Requirements for 11 Jun 87

Simulation of the System

To truly test the performance of the Server Scheduler, simulations were constructed for two typical days from the three-week period. The days chosen were 6 Jun 87 and 11 Jun 87. Four different simulations were run for each day, each using different assumptions. A summary of the assumptions for each of the five simulations follows:

1. All parameters were assumed to be deterministic. Customers were assumed to arrive at a constant rate throughout each hour, and the service time for each customer was assumed to be 5.156 minutes.

2. Customer interarrival times are distributed exponentially with a known mean. Service times are also randomly distributed, with the distribution given by Eq (24).

3. Customer interarrival times are distributed exponentially, but now the mean interarrival time is a random variable with a normal distribution. The mean of the mean interarrival time is known, and the standard deviation of the mean interarrival time is 5% of the mean. Service time distributions are still given by Eq (24).

4. Same as case 3, but now the standard deviation of the mean interarrival time is 10% of the mean.

The simulations were run using SLAM on a DEC VAX-8650 running under the VMS operating system. The SLAM model is shown in Figure 8, and the SLAM code and output for each of the 8 runs is given in Appendix C. The customer waiting time during each period was averaged across 50 runs of each simulation. A plot of this average is given in

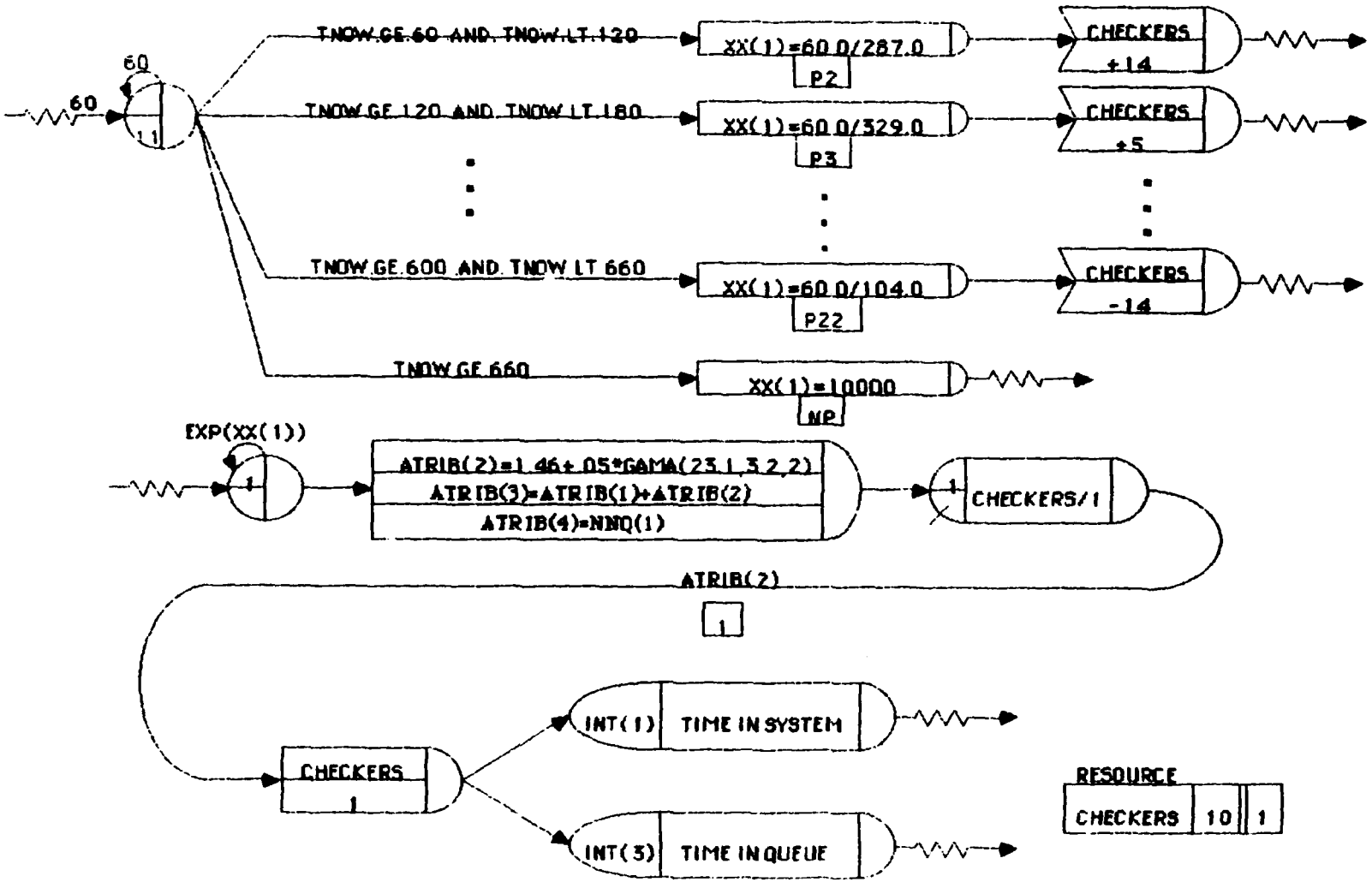


Figure 8 SLAM Model

Figures 11 to 18 for each simulation, along with the expected results according to the fluid approximation used in the Server Scheduler (Figures 9 and 10).

Comparison of the simulation results against the expected results according to the fluid approximation of the Server Scheduler shows that for some cases, the Server Scheduler is fairly accurate, and for other cases, the Server Scheduler is very inaccurate. Specifically, the Server Scheduler produces good results early in the day. However, toward the end of each day, the customer waiting time in the simulations exhibited a marked increase. Two approximations were made in the Server Scheduler that might have caused this problem. They were the approximations used to calculate the number of customers at the end of each period and to calculate the mean customer waiting time for each period.

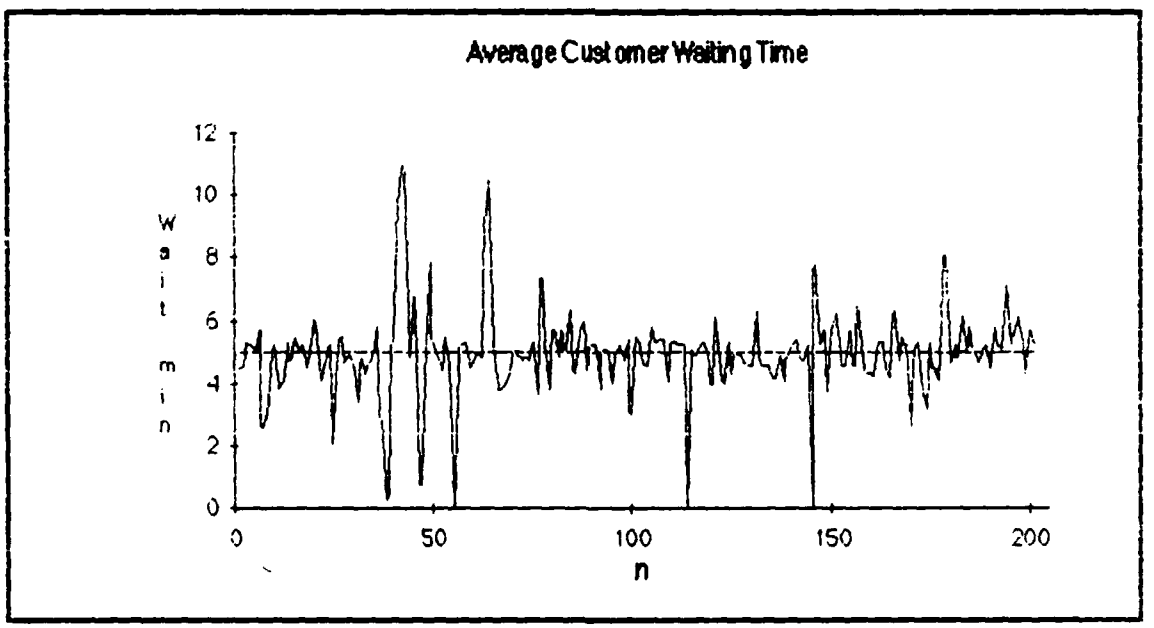


Figure 9 Fluid Approximation for Entire Scheduling Period

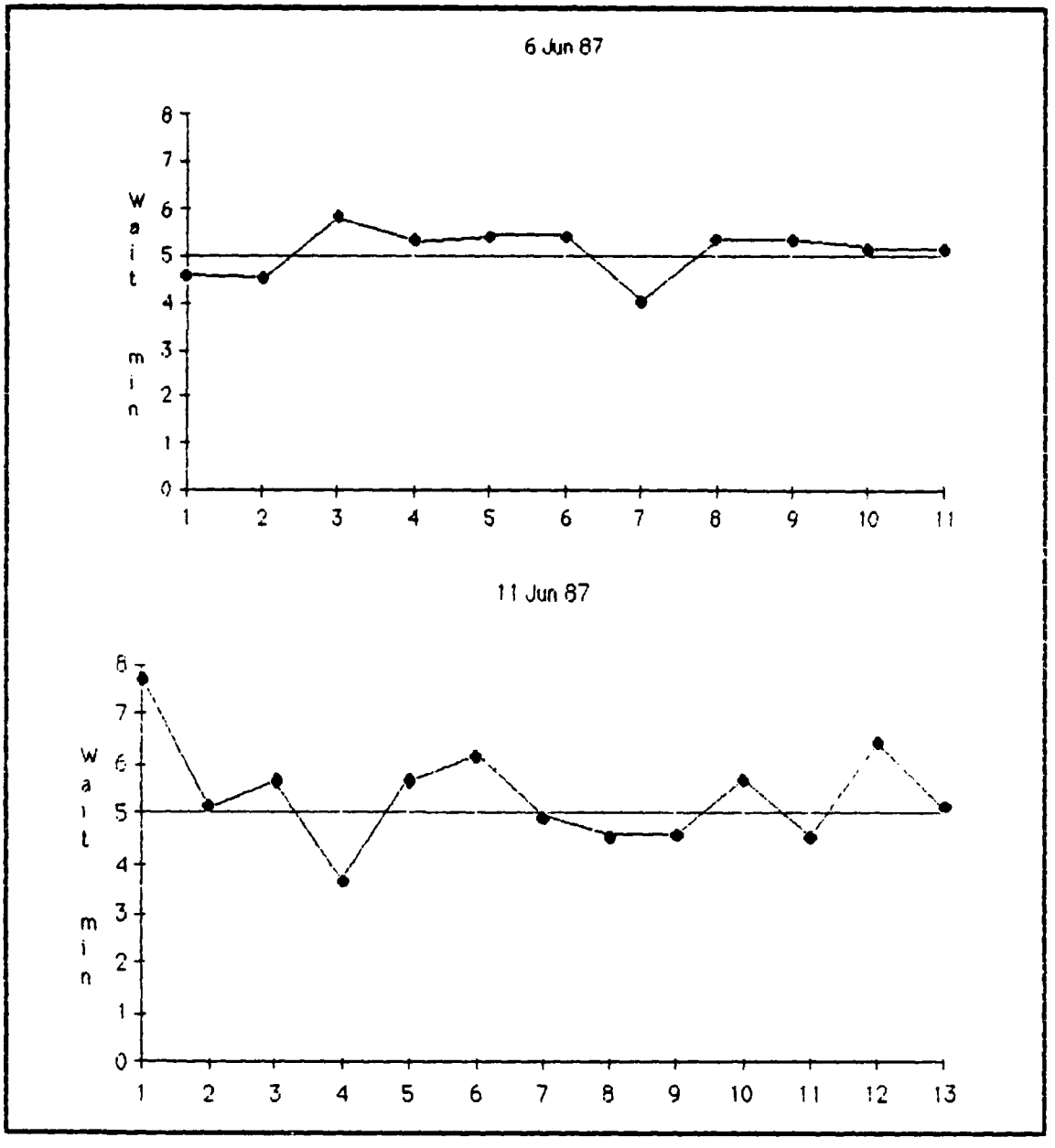


Figure 10 Fluid Approximation for 6 and 11 Jun 87

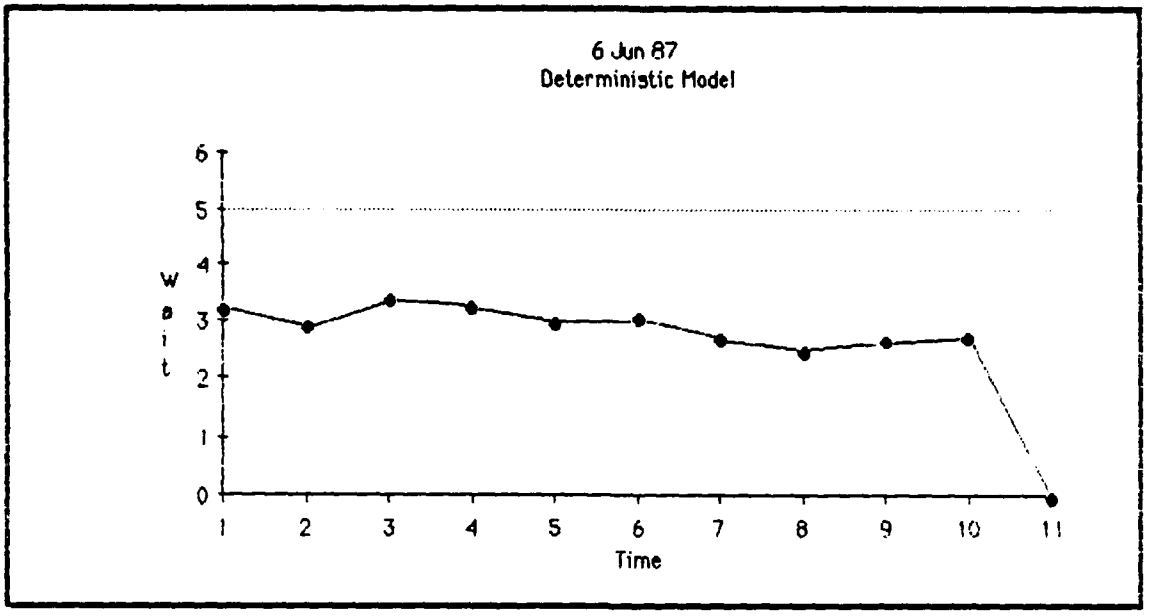


Figure 11 Case 1: 6 Jun 87

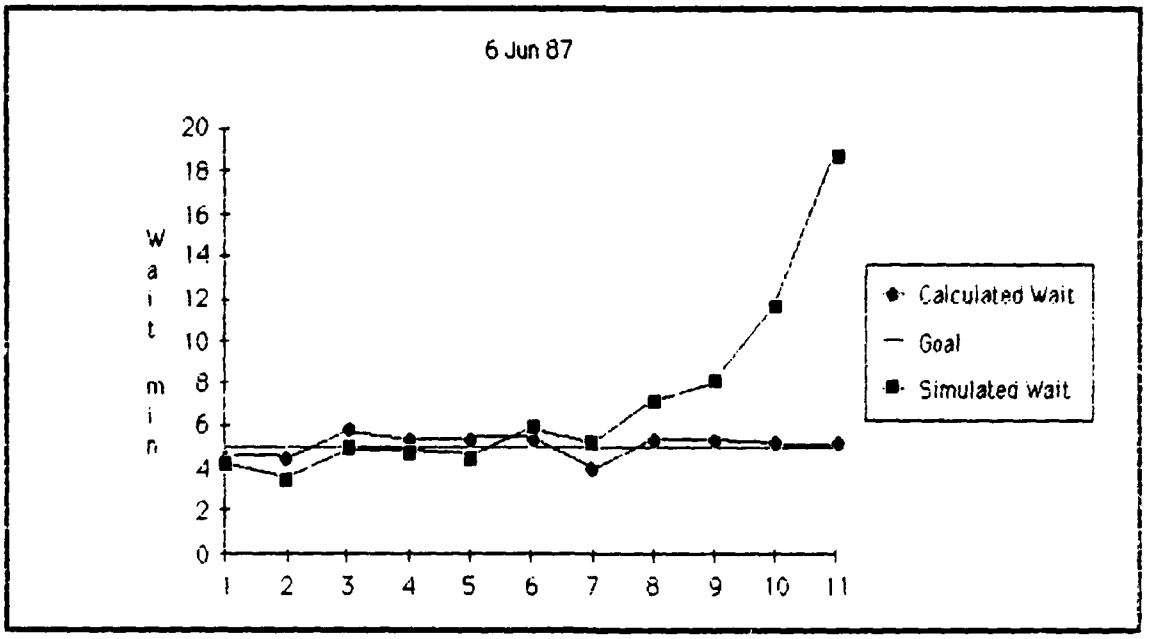


Figure 12 Case 2: 6 Jun 87

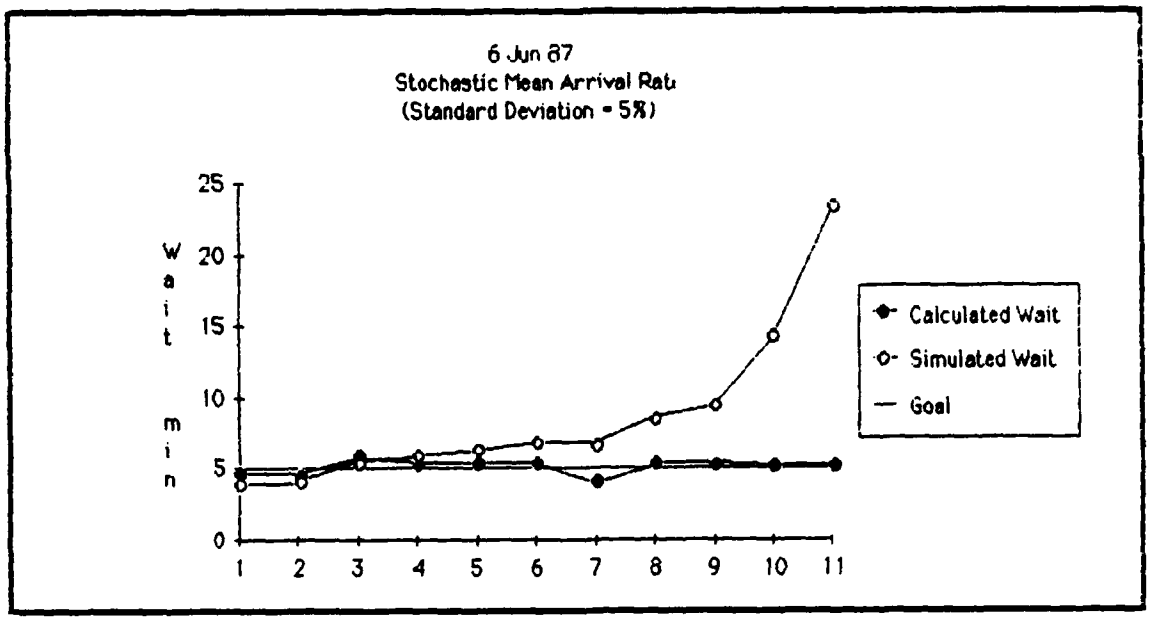


Figure 13 Case 3: 6 Jun 87

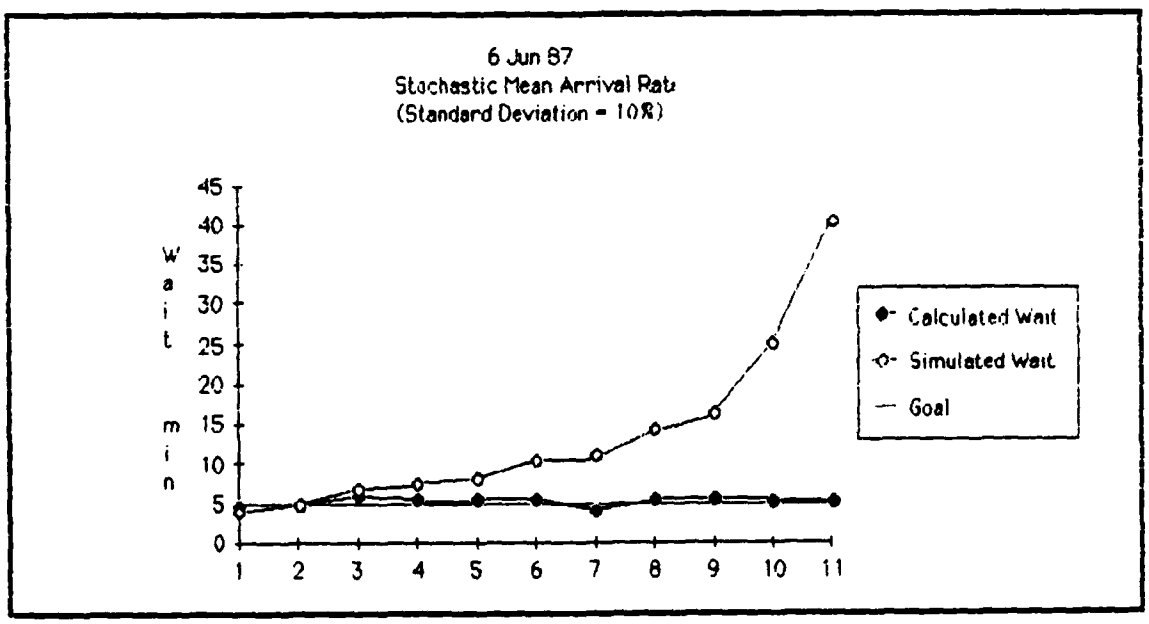


Figure 14 Case 4: 6 Jun 87

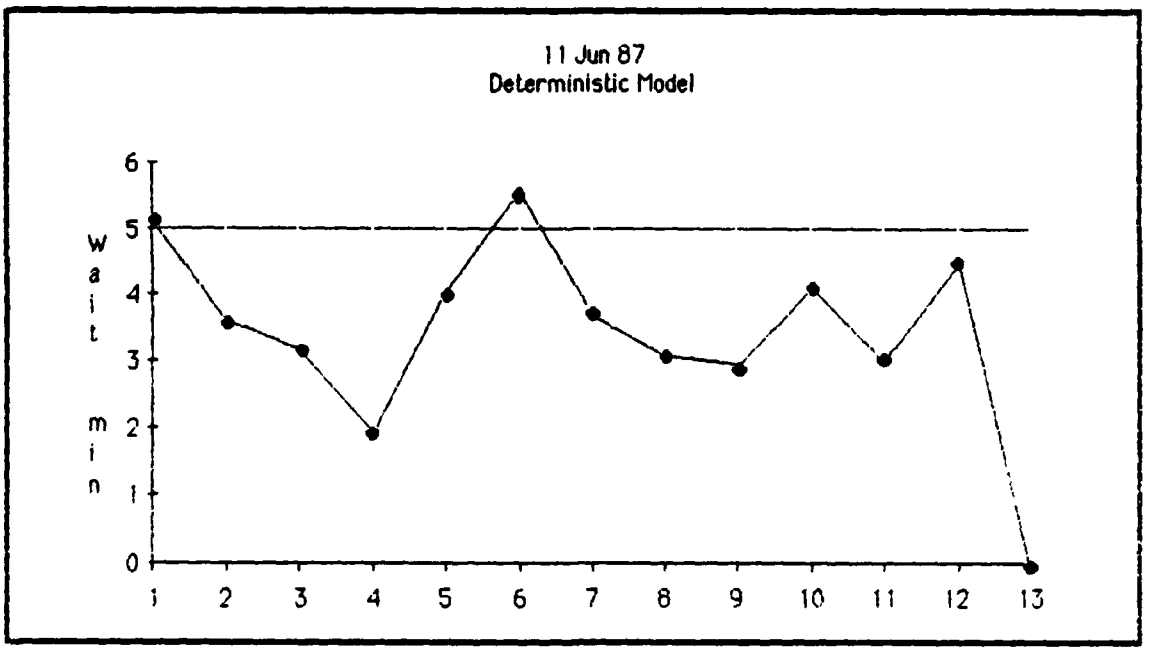


Figure 15 Case 1: 11 Jun 87

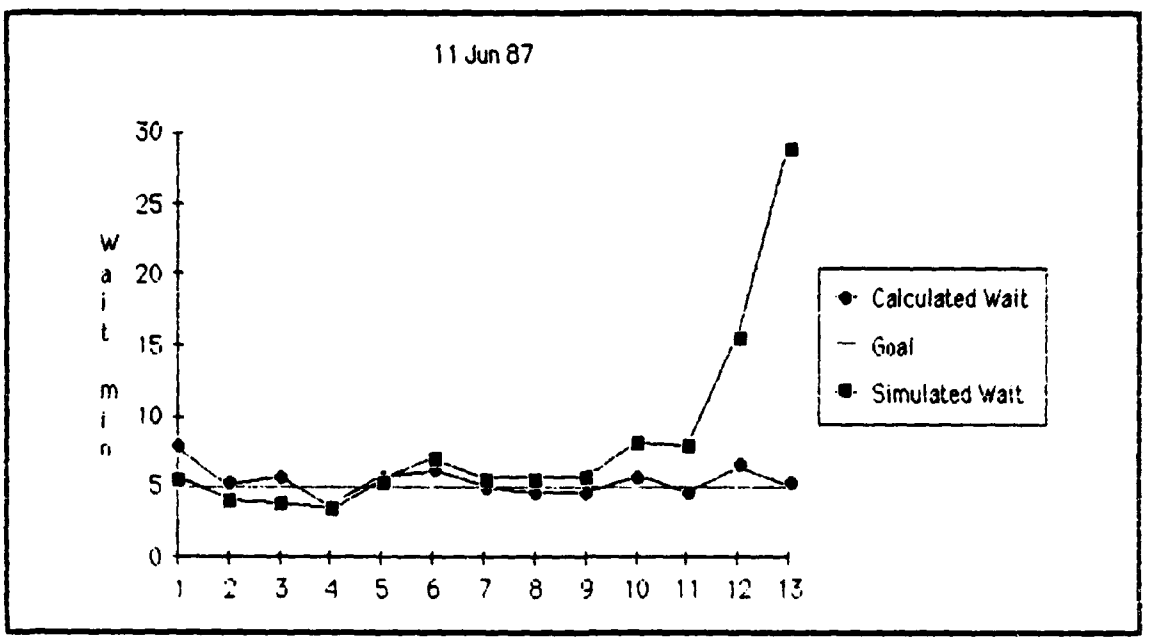


Figure 16 Case 2: 11 Jun 87

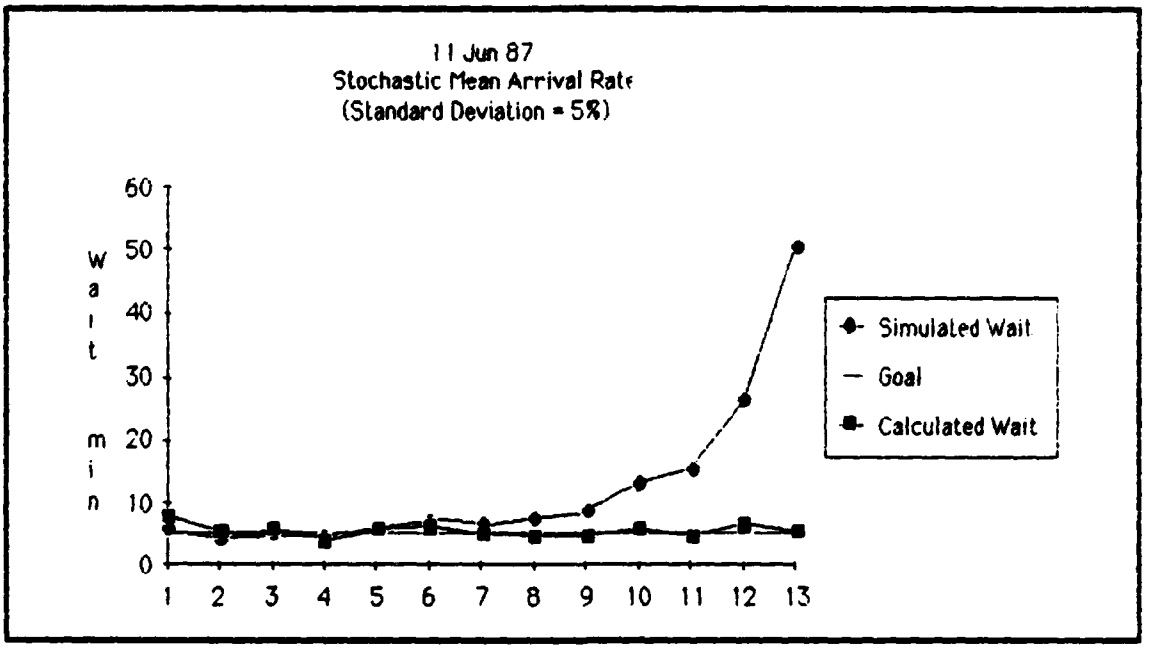


Figure 17 Case 3: 11 Jun 87

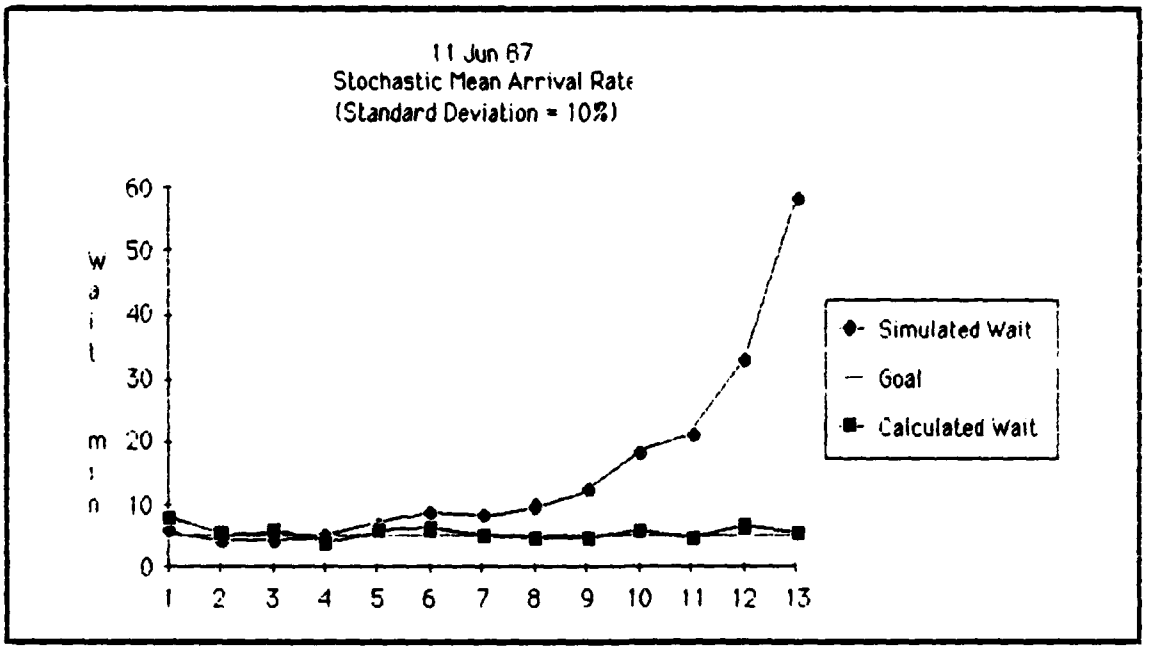


Figure 18 Case 4: 11 Jun 87

The approximation for the mean customer waiting time of each period is given by Eq (28):

$$W_n = \frac{L_n + L_{n+1}}{2\mu k_n} \qquad (28)$$

To test the accuracy of this approximation, the Case 2 simulation for 11 Jun 87 was repeated, and the number of customers at the end of each period was averaged across 50 simulation runs. This allows a comparison between the mean customer waiting times of each period and the approximation of Eq (28) using actual simulation values for L_n and L_{n+1}. This comparison is made in Figure 19, and the approximation appears to be fairly accurate.

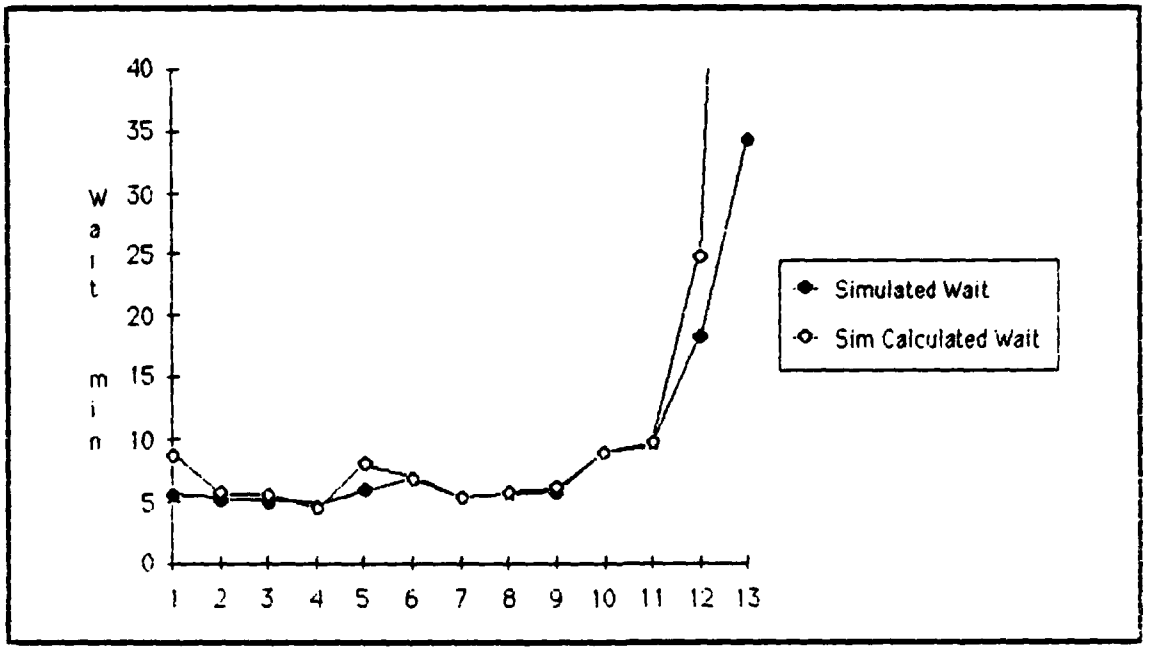


Figure 19 Comparison of Approximation and Actual Customer Waiting Times

The approximation for number of customers in line at the end of each period is given by Eq (29):

$$L_{n+1} = \begin{cases} L_n + A_n - 60\mu k_n & \text{if } n \neq \text{final hour of day} \\ 0 & \text{if } n = \text{final hour of day} \end{cases} \quad (29)$$

The values of L_n from the last simulation are compared to the expected values of L_n given by the approximation of Eq (29) in Figure 20. Notice that near the end of the day the approximation for L_n becomes inaccurate. This is the probable cause of the long waiting times observed at the end of each day in the simulations.

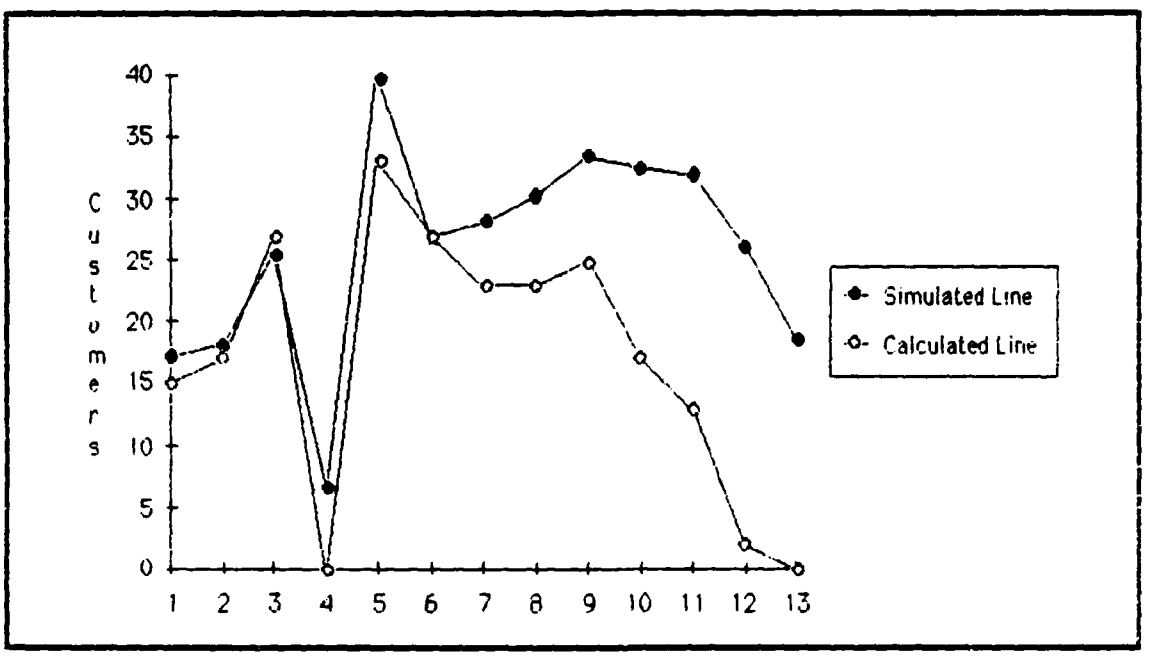


Figure 20 Comparison of Approximation and Actual Customer Line Lengths

To correct for the end of the day underestimation of line lengths show in Figure 20, an additional checker can be added at period 10. The effect of the additional checker, shown in Figure 21, is dramatic. The mean customer waiting time is reduced to acceptable levels.

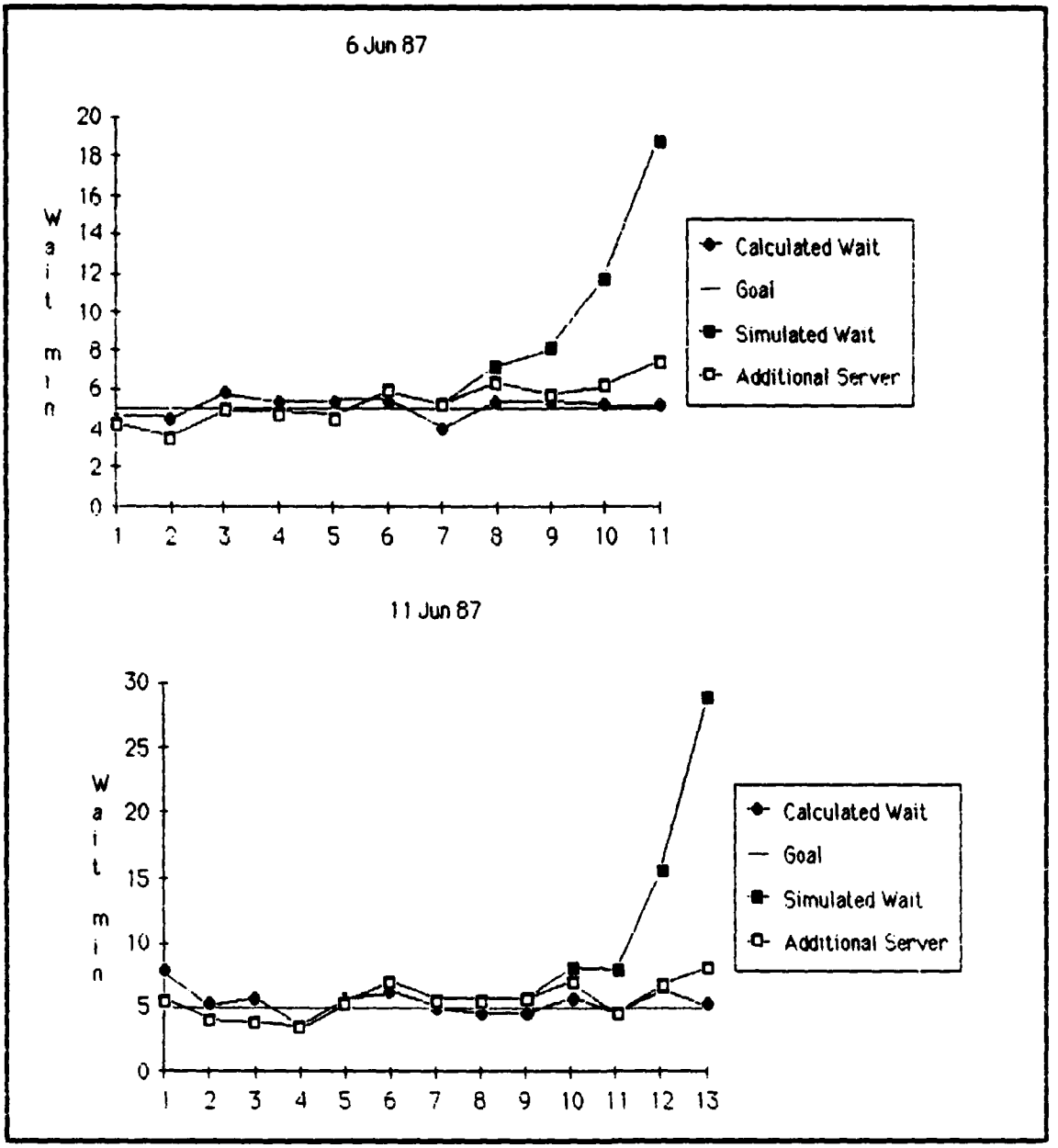


Figure 21 Effect of Additional Server at End of Day

V. Results, Recommendations, and Conclusions

Summary of Results

The primary objective of this research was to develop an analytical model that could be used to optimally schedule commissary checkers so that the expected customer waiting time remains relatively constant throughout the day. Such a model was developed and proved valid for most of the day. There were slight problems with the model at the end of each day, but these could be corrected by heuristically adding a server when the customer waiting time begins to increase. Since commissary managers typically have a number of discretionary employees available for temporary surges, this model could work in practice.

A subobjective was to make the model sufficiently general to be used in any Air Force commissary. The model developed is actually general enough to be used in any service organization where the customer arrival pattern is known. It allows the user to specify a mean customer waiting time goal, the second subobjective of the research. The deallocation procedure ensures that the third subobjective, to keep the total checker-hours below a maximum allowable number, is met.

The checker scheduling model was validated using simulation. The simulation showed that the model worked for most of the scheduling period. However, as mentioned previously, the model did tend to produce overly optimistic estimates of mean customer waiting times late in the day.

Recommendations

The Server Scheduler program could be a valuable tool for scheduling commissary checkers. The program is overly optimistic toward

the end of each day, but this problem can be corrected if only a single discretionary employee is available for one scheduling period near the end of the day. To eliminate the need for discretionary employees, a different approximation could be investigated that can successfully predict customer line lengths as they approach zero.

The Server Scheduler program implements two phases of a three phase problem. Although all objectives of the research were met, there are two obvious extensions of this work. The first is to solve the third phase of the problem, which is the assignment of actual people to specific shifts. The second is to improve the Phase II part of the algorithm (where the number of checkers to assign to each shift is determined). Currently, the Server Scheduler does not take lunch breaks into account. Several possible techniques for accomplishing this were outlined in Chapter III, along with limitations of these techniques.

Conclusions

The Server Scheduler provides a reliable and automated method for scheduling commissary checkers. The method can save management time, and it can save the commissary money through improved utilization of checkers. The method is sufficiently versatile to be used in any commissary in the Air Force.

Appendix A: Server Scheduler

```
program Server_Scheduler;
{
    Phase I of this program calculates the optimal number of
    checkers needed throughout the day (month) in order to achieve
    a five minute customer waiting time.

    Phase II of this program then determines the optimal number of
    checkers to assign to each shift in order to meet the Phase I
    checker requirements.
}

uses PasPrinter;

{*************************************************************************}

const
  MAXSTAGES = 372;                              { max # stages in a month }
  MU = 0.19395;                                    { average service rate }
  MUINV = 5.156;                                   { average service time }
  TN = 60.0;                                          { time in one stage }
  MAX_CHECKERS = 30;                    { maximum number of open checkers }
  MIN_CHECKERS = 1;                     { minimum number of open checkers }
  LMAX = 100;                         { maximum number of customers in line }
  INFINITY = 3.4E38;                { biggest allowed real number = infinity }
  INT_INFINITY = 2000000000;          { biggest allowed integer = infinity }
  MaxArcs = 255;          { maximum # arcs allowed in MinCostFlow procedure }
  MaxNodes = 254;        { maximum # nodes allowed in MinCostFlow procedure }

type
  IntArray1 = array[0..LMAX] of integer;
  RealArray1 = array[0..LMAX] of real;
  IntArray2 = array[1..MAXSTAGES] of integer;
  DayOfWeek = (Monday,Tuesday,Wednesday,Thursday,Friday,Saturday,Sunday);
  Hours = record
     open : integer;
     close : integer;
    end;
  date = record
     month : 1..12;
     day : 1..31;
     year : 1980..2100;
    end;
  ArcArray = array[1..MaxArcs] of integer;
  NodeArray = array[1..MaxNodes] of integer;

var
  n : integer;                                          { stage variable }
  hour : integer;                          { each hourly interval in a day }
  MM : integer;                               { total # stages in a month }
  TotalStages : integer;                    { also total # stages in a month }
  TotalHours : integer;                    { total # checker hours allocated }
  MaxHours : longint;            { maximum # checker hours to be allocated }
  k : IntArray2;                        { number of checkers open in stage n }
  d : array[1..MAXSTAGES] of ^IntArray1;          { optimal decision table }
  f : array[1..MAXSTAGES] of ^RealArray1;       { forward recursion values }
```

```
  L : integer;                              { number of customers in line }
  next_L : integer;            { number of customers in line at next stage }
  A : IntArray2;                            { number of arrivals in stage n }
  w : real;                                        { customer waiting time }
  target : real;                           { desired customer waiting time }
  temp1 : integer;                                   { temporary variable }
  temp2 : real;                                      { temporary variable }
  temp3 : real;                                      { temporary variable }
  R : real;                                       { return function value }
  i,j : integer;                          { integer loop control variables }
  dep : integer;                  { number of customer departures in stage n }
  time : array[DayOfWeek] of hours;              { daily hours of the store }
  CurrentDay : DayOfWeek;                         { current day of the week }
  FirstDay : DayOfWeek;                    { 1st day of scheduling period }
  LastDay : DayOfWeek;                     { last day of scheduling period }
  choice : integer;    { used in menu to select portion of program to run }
  done : boolean;                                  { terminates the program }
  FirstDate : date;                        { 1st date of scheduling period }
  LastDate : date;                         { last date of scheduling period }
  CurrentDate : date;                 { current date in scheduling period }
  ExceedHours : boolean;                 { tells if MaxHours was exceeded }

{*************************************************************************}

function calc_num_cust (last_num_cust,arrivals,num_servers :
                  integer; interval,service_rate : real) : integer;
{
    This function calculates the number of customers in line at
    the end of the next stage given:

        last_num_cust = # customers at the beginning of the stage
        arrivals = # customers arriving during the stage
        num_servers = # servers open during the stage
        service_rate = mean service rate of all open servers
        interval = length of time in the stage
}
var
  temp : integer;       { store # customers to check if it is nonnegative }

begin
  temp := last_num_cust + arrivals - trunc(interval*service_rate*num_servers);
  if temp < 0 then
    calc_num_cust := 0
  else
    calc_num_cust := temp;
end;

{*************************************************************************}

function waiting_time (num_cust,next_num_cust,num_servers :
                     integer; service_time : real) : real;
{
    This function calculates the mean customer waiting time in time
    period n given:

        num_cust = number of customers in line at the beginning of
                   time period n
        next_num_cust = number of customers in line at the end of
```

```
                        time period n
        num_servers = number of servers open during time period n
        service_time = mean service time of all servers
}
begin
  waiting_time := (num_cust + next_num_cust) * 0.5 * service_time / num_servers;
end;

{*************************************************************************}

function Return (Wait,desired_wait : real) : real;
{
    This function is the return function of the dynamic programming
    formulation:
                    R[n] = | Wait[n] - desired_wait |
    where:
        Wait = the mean customer waiting time
        desired_wait = desired customer waiting time
}
begin
  Return := Abs(Wait - desired_wait);
end;

{*************************************************************************}

procedure Deallocate;
{
     If the dynamic programming algorithm results in an optimal
     allocation of checkers which is above the limit on total
     hours, this procedure is used to remove checkers from stages
     so that the change in the return function is minimized.
}

var
  stage : integer;
  best_place : integer;                    { best stage to remove a server }
  z : real;                    { current best value of objective function }
  new_z : real;                { value of objective function to be tested }

begin


  while (TotalHours > MaxHours) do
    begin
      z := INFINITY;
      for n := 1 to NN do            { find best place to remove a server }
        begin
          if k[n] > MIN_CHECKERS then
            begin
              k[n] := k[n] - 1;     { reduce number of servers at stage n }

            { Calculate the new return }

              CurrentDay := FirstDay;
              hour := time[CurrentDay].open;
              new_z := 0;
              L := 0;
```

```
            for stage := 1 to NN do
              begin
                if (hour = (time[CurrentDay].close - int(TN/60.0*100))) then
                  next_L := 0
                else
                  next_L := calc_num_cust(L,A[stage],k[stage],TN,MU);
                w := waiting_time(L,next_L,k[stage],MUINU);
                R := Return(w,TARGET);
                new_z := new_z + R;

                if (hour = (time[CurrentDay].close - int(TN/60.0*100))) then
                  begin
                    if CurrentDay = Sunday then
                      CurrentDay := Monday
                    else
                      CurrentDay := succ(CurrentDay);
                    hour := time[CurrentDay].open;
                  end
                else
                  hour := hour + int(TN/60.0*100);
              end; { for stage := 1 to NN }

            { if the new return is better, remember it }

              if new_z < z then
                begin
                  z := new_z;
                  best_place := n;
                end; { if new_z < z }

              k[n] := k[n] + 1;   { restore number of servers at stage n }

            end; { if k[n] > MIN_CHECKERS }

          end; { for n := 1 to NN }

        { Remove server at the best place }

          k[best_place] := k[best_place] - 1;

      end; { while (TotalHours > MaxHours) }

  end; { procedure Deallocate }


  {*************************************************************}

  procedure FillDPTable;
  var
    ch : char;

  begin

    ClearScreen;
    writeln('Starting to fill DP state-stage table');

    CurrentDay := LastDay;
    if time[CurrentDay].open = time[CurrentDay].close then
```

```
  begin
    if CurrentDay = Monday then
      CurrentDay := Sunday
    else
      CurrentDay := pred(CurrentDay);
  end;


hour := time[CurrentDay].close;          { Begin with last stage }
n := NN;
writeln('n = ',NN);
for L := 0 to LMAX do
  begin
    temp2 := 100000.0;
    if ((TN * MU * MAX_CHECKERS) < L) then
      begin
        temp1 := MAX_CHECKERS;
        w := waiting_time(L,0,MAX_CHECKERS,MUINV);
        R := Return(w,target);
        temp2 := R;
      end { if ((TN * MU * MAX_CHECKERS) < L) }
    else
      begin
        for j := MIN_CHECKERS to MAX_CHECKERS do
          begin
            if ((TN * MU * j) >= (L + A[NN])) then
              begin
                w := waiting_time(L,0,j,MUINV);
                R := Return(w,target);
                if R < temp2 then
                  begin
                    temp2 := R;
                    temp1 := j;
                  end; { if R < temp2 }
              end; { if ((TN * MU * j) >= (L+A[NN])) }
          end; { j := MIN_CHECKERS to MAX_CHECKERS }
      end; { else }
    d[NN]^[L] := temp1;
    f[NN]^[L] := temp2;
  end; { for L := 0 to LMAX }
hour := hour - int(TN/60.0*100);

for n := NN-1 downto 1 do
  begin
    writeln('n = ',n);
    if (hour = time[CurrentDay].open) then
      begin
        temp2 := 100000.0;
        for j := MIN_CHECKERS to MAX_CHECKERS do
          begin
            next_L := calc_num_cust(0,A[n],j,TN,MU);
            if next_L < LMAX then
              begin
                w := waiting_time(0,next_L,j,MUINV);
                R := Return(w,target);
                if (R+f[n+1]^[next_L]) < temp2 then
                  begin
                    temp2 := R+f[n+1]^[next_L];
```

```
                temp1 := j;
              end;
          end; { if next_L < LMAX then }
      end; { j := MIN_CHECKERS to MAX_CHECKERS }
    d[n]^[0] := temp1;
    f[n]^[0] := temp2;
  end { if hour = time[CurrentDay].open }
else
  begin
    if (hour = (time[CurrentDay].close)) then
      begin
        for L := 0 to LMAX do
          begin
            temp2 := 100000.0;
            if ((TN * MU * MAX_CHECKERS) < (L + A[n])) then
              begin
                temp1 := MAX_CHECKERS;
                next_L := calc_num_cust(L,A[n],MAX_CHECKERS,TN,MU);
                w := waiting_time(L,next_L,MAX_CHECKERS,MUINV);
                R := Return(w,target);
                temp2 := R + f[n+1]^[0];
              end { if ((TN * MU * MAX_CHECKERS) < (L + A[n])) }
            else
              begin
                for j := MIN_CHECKERS to MAX_CHECKERS do
                  begin
                    if ((TN * MU * j) >= (L + A[n])) then
                      begin
                        w := waiting_time(L,0,j,MUINV);
                        R := Return(w,target);
                        if ((R + f[n+1]^[0]) < temp2) then
                          begin
                            temp2 := R + f[n+1]^[0];
                            temp1 := j;
                          end; { if R + f[n+1]^[0] < temp2 }
                      end; { if ((TN * MU * j) >= L) }
                  end; { j := MIN_CHECKERS to MAX_CHECKERS }
              end; { else }
            d[n]^[L] := temp1;
            f[n]^[L] := temp2;
          end; { for L := 0 to LMAX }
      end
    else { normal hour of the day }
      begin
        for L := 0 to LMAX do
          begin
            temp2 := 100000.0;
            for j := MIN_CHECKERS to MAX_CHECKERS do
              begin
                next_L := calc_num_cust(L,A[n],j,TN,MU);
                if next_L < LMAX then
                  begin
                    w := waiting_time(L,next_L,j,MUINV);
                    R := Return(w,target);
                    if (R+f[n+1]^[next_L]) < temp2 then
                      begin
                        temp2 := R+f[n+1]^[next_L];
                        temp1 := j;
```

```
                        end; { if (R+f[n+1]^[next_L]) < temp2 }
                    end; { if next_L < LMAX }
                end; { j := MIN_CHECKERS to MAX_CHECKERS }
                d[n]^[L] := temp1;
                f[n]^[L] := temp2;
              end; { L := 0 to LMAX }
          end; { else }
      end; { else }

    if (hour = time[CurrentDay].open) then
      begin
        if CurrentDay = Monday then
          CurrentDay := Sunday
        else
          CurrentDay := Pred(CurrentDay);
        if time[CurrentDay].open = time[CurrentDay].close then
          begin
            if CurrentDay = Monday then
              CurrentDay := Sunday
            else
              CurrentDay := Pred(CurrentDay);
          end;
        hour := time[CurrentDay].close;
      end { if (hour = time[CurrentDay].open) }
    else
      hour := hour - int(TN/60.0*100);

  end; { n := NN-1 downto 1 }

end; { procedure FillDPTable }

{*************************************************************************}

procedure ForwardPass;
var
  ch : char;
begin

  writeln('Starting forward pass in DP');

  L := 0;
  CurrentDay := FirstDay;
  if time[CurrentDay].open = time[CurrentDay].close then
    begin
      if CurrentDay = Sunday then
        CurrentDay := Monday
      else
        CurrentDay := succ(CurrentDay);
    end;
  hour := time[CurrentDay].open;

  TotalHours := 0;

  for n := 1 to NN do
    begin
      writeln('n = ',n);
      k[n] := d[n]^[L];
      TotalHours := TotalHours + k[n];
```

```
        if (hour = (time[CurrentDay].close)) then
          next_L := 0
        else
          next_L := calc_num_cust(L,A[n],k[n],TN,MU);
        L := next_L;
        if (hour = time[CurrentDay].close) then
          begin
            if (CurrentDay = Sunday) then
              CurrentDay := Monday
            else
              CurrentDay := Succ(CurrentDay);
            if time[CurrentDay].open = time[CurrentDay].close then
              begin
                if CurrentDay = Sunday then
                  CurrentDay := Monday
                else
                  CurrentDay := succ(CurrentDay);
              end;
            hour := time[CurrentDay].open;
          end { if (hour = time[CurrentDay].close) }
        else
          hour := hour + int(TN/60.0*100);
      end; { n := 1 to NN }
end;{ procedure ForwardPass }

{*************************************************************************}

procedure GetArrivals(var NumStages:integer; var arrivals:IntArray2);

var
  InputFile : text;
  n : integer;

begin
  reset(InputFile,'Arrivals.dat');
  n := 0;
  while not SeekEof(InputFile) do
    begin
      n := n + 1;
      if not SeekEoln(InputFile) then
        read(InputFile,arrivals[n])
      else
        begin
          readln;
          read(InputFile,arrivals[n])
        end;
    end;
  Close(InputFile);
  NumStages := n;
end;

{*************************************************************************}

procedure InitHours;
{
     This procedure sets default store hours as:
```

```
                Day          Open         Close
              Monday         0000          0000
              Tuesday        0900          2000
              Wednesday      0900          1900
              Thursday       0900          2100
              Friday         0900          1900
              Saturday       0800          1800
              Sunday         0900          1700
}

var
  day : DayOfWeek;

begin
  time[Monday].open := 0000;
  time[Monday].close := 0000;
  time[Tuesday].open := 0900;
  time[Tuesday].close := 2000;
  time[Wednesday].open := 0900;
  time[Wednesday].close := 1900;
  time[Thursday].open := 0900;
  time[Thursday].close := 2100;
  time[Friday].open := 0900;
  time[Friday].close := 1900;
  time[Saturday].open := 0800;
  time[Saturday].close := 1800;
  time[Sunday].open := 0900;
  time[Sunday].close := 1700;
end;  { procedure InitHours }

{*****************************************************************}

procedure InitPointers(number_of_stages : integer);
{
     This procedure initializes the pointers for dynamic memory
     allocation.
}
begin
  for n := 1 to number_of_stages do
    begin
      New(d[n]);
      New(f[n]);
    end;
end;

{*****************************************************************}

procedure InitTarget;
{
     This procedure sets the default customer waiting time goal
     to 5.0 minutes.
}
begin
  target := 5.0;
end; { procedure InitTarget }

{*****************************************************************}
```

```
procedure InitMaxHours;
{
     This procedure sets the default maximum allowable checker hours
     to infinity.
}
begin
  MaxHours := INT_INFINITY;
end; { procedure InitMaxHours }

{*************************************************************************}

procedure Menu(var selection : integer);

begin
  ClearScreen;
  GotoXY(23,10);
  write('1. Run Phase I');
  GotoXY(23,11);
  write('2. Run Phase II');
  GotoXY(23,12);
  write('3. Run Phase I & Phase II');
  GotoXY(23,13);
  write('4. Set Weekly Store Operating Hours');
  GotoXY(23,14);
  write('5. Set Desired Customer Waiting Time');
  GotoXY(23,15);
  write('6. Set Limit on Total Checker Hours');
  GotoXY(23,16);
  write('7. Quit');
  GotoXY(23,19);
  write('Enter Selection:');
  readln(selection);
  ClearScreen;
end;

{*************************************************************************}

procedure ReadDate(var FirstDate, LastDate : date);
{
     This procedure asks the user for the first and last day to
     be scheduled and converts the input into the proper date format.
}

var
  FirstDay, LastDay : string[10];
  BadResponse : boolean;

begin
  ClearScreen;
  repeat
    BadResponse := false;
    GotoXY(12,12);
    write('Enter first day of scheduling period (mm/dd/yyyy):');
    readln(FirstDay);
    if (ord(FirstDay[1]) > 49) or (ord(FirstDay[1]) < 48) then
      BadResponse := true;
    if (ord(FirstDay[2]) > 57) or (ord(FirstDay[2]) < 48) then
      BadResponse := true;
```

```
    if (FirstDay[1] = '1') and (ord(FirstDay[2]) > 50) then
      BadResponse := true;
    if (ord(FirstDay[4]) > 51) or (ord(FirstDay[4]) < 48) then
      BadResponse := true;
    if (ord(FirstDay[5]) > 57) or (ord(FirstDay[5]) < 48) then
      BadResponse := true;
    if (ord(FirstDay[7]) > 50) or (ord(FirstDay[7]) < 49) then
      BadResponse := true;
    if (ord(FirstDay[8]) > 57) or (ord(FirstDay[8]) < 48) then
      BadResponse := true;
    if (ord(FirstDay[9]) > 57) or (ord(FirstDay[9]) < 48) then
      BadResponse := true;
    if (ord(FirstDay[10]) > 57) or (ord(FirstDay[10]) < 48) then
      BadResponse := true;
    if (FirstDay[3] <> '/') or (FirstDay[6] <> '/') then
      BadResponse := true;
    if BadResponse then
      begin
        ClearScreen;
        GotoXY(12,10);
        write('Incorrect format or out of possible range, try again.');
      end
    else
      begin
        GotoXY(12,10);
        ClearEOL;
      end;
  until not BadResponse;
  repeat
    BadResponse := false;
    GotoXY(12,13);
    write('Enter last day of scheduling period (mm/dd/yyyy):');
    readln(LastDay);
    if (ord(LastDay[1]) > 49) or (ord(LastDay[1]) < 48) then
      BadResponse := true;
    if (ord(LastDay[2]) > 57) or (ord(LastDay[2]) < 48) then
      BadResponse := true;
    if (LastDay[1] = '1') and (ord(LastDay[2]) > 50) then
      BadResponse := true;
    if (ord(LastDay[4]) > 51) or (ord(LastDay[4]) < 48) then
      BadResponse := true;
    if (ord(LastDay[5]) > 57) or (ord(LastDay[5]) < 48) then
      BadResponse := true;
    if (ord(LastDay[7]) > 50) or (ord(LastDay[7]) < 48) then
      BadResponse := true;
    if (ord(LastDay[8]) > 57) or (ord(LastDay[8]) < 48) then
      BadResponse := true;
    if (ord(LastDay[9]) > 57) or (ord(LastDay[9]) < 48) then
      BadResponse := true;
    if (ord(LastDay[10]) > 57) or (ord(LastDay[10]) < 48) then
      BadResponse := true;
    if (LastDay[3] <> '/') or (LastDay[6] <> '/') then
      BadResponse := true;
    if BadResponse then
      begin
        GotoXY(61,13);
        ClearEOL;
        GotoXY(12,10);
```

```
        write('Incorrect format or out of possible range, try again.');
      end
    else
      begin
        GotoXY(12,10);
        ClearEOL;
      end;
  until not BadResponse;

{ Convert date from string format to 'date' format (integer) }

  FirstDate.month := 10 * (ord(FirstDay[1]) - 48) + (ord(FirstDay[2]) - 48);
  FirstDate.day := 10 * (ord(FirstDay[4]) - 48) + (ord(FirstDay[5]) - 48);
  FirstDate.year := 1000 * (ord(FirstDay[7]) - 48) + 100 * (ord(FirstDay[8]) -
48)
          + 10 * (ord(FirstDay[9]) - 48) + (ord(FirstDay[10]) - 48);
  LastDate.month := 10 * (ord(LastDay[1]) - 48) + (ord(LastDay[2]) - 48);
  LastDate.day := 10 * (ord(LastDay[4]) - 48) + (ord(LastDay[5]) - 48);
  LastDate.year := 1000 * (ord(LastDay[7]) - 48) + 100 * (ord(LastDay[8]) - 48)
          + 10 * (ord(LastDay[9]) - 48) + (ord(LastDay[10]) - 48);

end; { procedure ReadDate }

{*********************************************************************}

function Julian(SomeDate : date) : integer;
{
     This function takes a date in 'standard' date format and
     converts it to Julian format ( yyyy and ddd )
}

var
  temp : integer;

begin
  case SomeDate.month of
    1  : temp := SomeDate.day;
    2  : temp := SomeDate.day + 31;
    3  : temp := SomeDate.day + 59;
    4  : temp := SomeDate.day + 90;
    5  : temp := SomeDate.day + 120;
    6  : temp := SomeDate.day + 151;
    7  : temp := SomeDate.day + 181;
    8  : temp := SomeDate.day + 212;
    9  : temp := SomeDate.day + 243;
    10 : temp := SomeDate.day + 273;
    11 : temp := SomeDate.day + 304;
    12 : temp := SomeDate.day + 334;
  end; { case SomeDate.month of }
  if ((SomeDate.year mod 4) = 0) then
    temp := temp + 1;
  Julian := temp;
end; { function Julian }

{*********************************************************************}

function DaysBetween(FirstDate,LastDate : date) : integer;
{
```

```
    This function calculates the number of days between two dates.
}

var
  i : integer;
  FirstJulian, LastJulian : integer;
  temp : integer;

begin
  FirstJulian := Julian(FirstDate);
  LastJulian := Julian(LastDate);
  temp := LastJulian - FirstJulian;
  for i := FirstDate.year to LastDate.year-1 do
    if (i mod 4) = 0 then
      temp := temp + 366
    else
      temp := temp + 365;
  DaysBetween := temp;
end; { function DaysBetween }

{*****************************************************************}

procedure SetDate(var StartDay,EndDay : DayOfWeek; var NumStages : integer);
{
    This procedure asks the user for the first and last day to
    be scheduled.  It then calculates the total number of stages
    (hours) in the scheduling period and the day of the week for the
    first and last days.
}

var
  NumDays,temp : integer;
  Ref : date;
  DayArray : array[0..6] of DayOfWeek;
  today : DayOfWeek;

begin
  Ref.month := 1;              { Reference date is Friday, 1 January 1988 }
  Ref.day := 1;
  Ref.year := 1988;
  DayArray[0] := Friday;
  DayArray[1] := Saturday;
  DayArray[2] := Sunday;
  DayArray[3] := Monday;
  DayArray[4] := Tuesday;
  DayArray[5] := Wednesday;
  DayArray[6] := Thursday;

  ReadDate(FirstDate,LastDate);

{ Determine the day of the week for the first day to be scheduled }
  NumDays := DaysBetween(Ref,FirstDate);
  temp := NumDays mod 7;
  StartDay := DayArray[temp];

{ Determine the day of the week for the last day to be scheduled }
  NumDays := DaysBetween(Ref,LastDate);
  temp := NumDays mod 7;
```

```
  EndDay := DayArray[temp];

  NumDays := DaysBetween(FirstDate,LastDate);
  today := StartDay;
  NumStages := 0;
  for temp := 0 to NumDays do
    begin
      if (time[today].open <> time[today].close) then
        NumStages := NumStages + time[today].close - time[today].open +
int(TN/60.0*100);
      if today = Sunday then
        today := Monday
      else
        today := Succ(today);
    end;
  NumStages := NumStages div 100;
  if TN <> 60 then
    NumStages := NumStages * round(60/TN);

end; { procedure SetDate }

{*********************************************************************}

procedure MinCostFlow(NumNodes,NumArcs : integer;
                      c : ArcArray;
                      var x : ArcArray;
                      F : ArcArray;
                      T : ArcArray;
                      b : NodeArray;
                      u : ArcArray);
{
     This procedure solves the Minimum Cost Flow problem:

                      Min      c'x
                      s.t.    Ax = b
                           0 ≤ x ≤ u

     where
       x = vector of arc flows
       c = vector of arc flow costs
       b = vector of node supply or demands
       u = vector of arc flow upper bounds
       A = node arc incidence matrix
       NumNodes = number of nodes in the network (max is 254)
       NumArcs = number of arcs in the network (max is 255 - NumNodes)

     Notice that the node arc incidence matrix was not one of the
     input parameters.  Instead, the arrays F & T are used, where:
          F = "From" function of an arc, i.e. F[x(i,j)] = i
          T = "To" function of an arc, i.e. T[x(i,j)] = j
     Notice that F & T require much less memory than A.

     The following constants must be defined before the procedure
     is called:
       MaxArcs = 255;
       MaxNodes = 254;

     The following types must be defined before the procedure
```

```
    is called:
      ArcArray = array[1..MaxArcs] of integer;
      NodeArray = array[1..MaxNodes] of integer;

    For added speed, all variables and parameters are defined as
    integers.  If noninteger values are needed, simply redefine
    the variables and parameters as reals.

    The out of kilter algorithm is used to solve the problem.
    Reference:  Algorithms for Network Programming, by Jeff L.
                Kennington & Richard V. Helgason. John Wiley
                & Sons, New York, 1980.
}

{*****************************************************************}

const
  inf = 9999;
type
  SlackNodeSet = set of 1..MaxNodes;
  SlackArcSet = set of 1..MaxArcs;

var
  pi : NodeArray;                                  { dual variables }
  cost : ArcArray;              { cost[j] = pi[F[j]] - pi[T[j]] - c[j] }
  theta : integer;                  { amount to change dual variables }
  psi1 : set of 1..MaxArcs;                  { candidate set for tree }
  psi2 : set of 1..MaxArcs;                  { candidate set for tree }
  Nhat : set of 1..MaxNodes;                  { current nodes in tree }
  Ahat : set of 1..MaxArcs;                    { current arcs in tree }
  delta : NodeArray;                    { amt to change flows in cycle }
  i : integer;                                 { loop control variable }
  j : integer;                                 { loop control variable }
  s : integer;                                    { out of kilter arc }
  need_arc : boolean; { indicates whether an out of kilter arc is known }
  in_kilter : boolean;       { indicates whether all arcs are in kilter }
  no_cycle : boolean;                { indicates whether a cycle exists }
  NumSlackArcs : integer;  { total # of arcs after slack arcs are added }
  NumSlackNodes : integer; { total # of nodes after slack node is added }

{*****************************************************************}

    procedure Initial_Solution;
    {
        This procedure finds an initial feasible solution to start
        the algorithm.

        The initial solution is found by use of the "all-artificial
        start."  An extra node, NumNodes+1, is added to the network.
        For each source node i with supply b[i], a slack arc is
        added from i to NumNodes+1 with c[slack arc] = 0,
        u[slack arc] = infinity, and x[slack arc] = b[i].
        For each demand node k with demand |b[k]|, a slack arc is
        added from NumNodes+1 to k with c[slack arc] = infinity,
        u[slack arc] = infinity, and x[slack arc] = |b[k]|.
    }

    var
```

```
    NL : set of 1..MaxNodes;
    NU : set of 1..MaxNodes;
    found : boolean;
    j : integer;
    i : integer;

  begin
    for j := 1 to NumArcs do
      begin
        x[j] := 0;
      end;
    for i := 1 to NumNodes do
      begin
        if b[i] > 0 then
          begin
            x[i+NumArcs] := b[i];
            c[i+NumArcs] := 0;
            u[i+NumArcs] := inf;
            T[i+NumArcs] := NumNodes + 1;
            F[i+NumArcs] := i;
          end; { if b[i] > 0 }
        if b[i] <= 0 then
          begin
            x[i+NumArcs] := -b[i];
            c[i+NumArcs] := inf;
            u[i+NumArcs] := inf;
            T[i+NumArcs] := i;
            F[i+NumArcs] := NumNodes + 1;
          end; { if b[i] <= 0 }
      end; { for i := 1 to NumNodes }

    for i := 1 to NumSlackNodes do        { fill pi[i] for all nodes }
      pi[i] := 0;

    for j := 1 to NumSlackArcs do     { compute cost[j] for all arcs }
      cost[j] := pi[F[j]] - pi[T[j]] - c[j];

  end; { procedure Initial_Solution }

{*****************************************************************}

  procedure ArcSearch(var in_kilter : boolean; var es : integer);

  {
      This procedure searchs for an out of kilter arc in the network.
      If all network arcs are in kilter, then the flag in_kilter is
      set to true.  Otherwise, the first out of kilter arc found is
      returned in es, and the flag in_kilter is set to false.

      This procedure assumes that all parameters are available, ie
      c[j], x[j], pi[n], & cost[j] = pi[F[j]] - pi[T[j]] - c[j]
  }
  begin
    in_kilter := true;
    j := 0;
    repeat
      j := j + 1;
      if (cost[j] < 0) and (x[j] > 0) then
```

```
            in_killer := false;
          if (cost[j] > 0) and (x[j] < u[j]) then
            in_killer := false;
        until (not in_killer) or (j = NumSlackArcs);
        if (not in_killer) then
          es := j;
      end; { procedure ArcSearch }

{*****************************************************************************}

      procedure primal(es : integer; var no_cycle_flag : boolean);
      {
           This procedure executes the primal phase of the out
           of killer algorithm with arc es.  If the algorithm
           terminates with the conclusion that no cycle exists,
            no_cycle_flag is set to true.  Otherwise,
            no_cycle_flag is set to false.
      }

      var
        found : boolean;
        j,i : integer;
        tree_path : ArcArray;
        num_path_arcs : integer;
        current_node : integer;

{*****************************************************************************}

          procedure find_path(k,l : integer; N : SlackNodeSet;
                              A : SlackArcSet;
                              var path : ArcArray;
                              var num_in_path : integer);
          {
               This procedure finds the path thru the tree N
               from node k to node l (assumes that there is
               only a single path in N from k to l).  The path
               is returned in the array path and the number of
               arcs in the path is returned in num_in_path.
          }
          var
            next_node : array[1..MaxNodes] of integer;
            found_nodes : SlackNodeSet;
            i,j, current_node : integer;
            found : boolean;

          begin
            next_node[l] := l;       { start at l - there's no next node }
            found_nodes := [ l ];
            repeat
              for j := 1 to NumSlackArcs do
                begin
                  if (j in A) then
                    begin
                      if (T[j] in found_nodes) and
                         (not(F[j] in found_nodes)) then
                        begin
                          found_nodes := found_nodes + [ F[j] ];
                          next_node[F[j]] := T[j];
```

```
                    end;
                  if (F[j] in found_nodes) and
                     (not(T[j] in found_nodes)) then
                    begin
                      found_nodes := found_nodes + [ T[j] ];
                      next_node[T[j]] := F[j];
                    end;
                end; { if (T[j] in N) and (F[j] in N) }
            end; { for j := 1 to NumSlackArcs }
          until (k in found_nodes);
          num_in_path := 0;
          current_node := k;
          repeat
            j := 0;
            found := false;
            while not found do
              begin
                j := j + 1;
                if ((j in A ) and (F[j] = next_node[current_node])
                     and (T[j] = current_node)) then
                  begin
                    num_in_path := num_in_path + 1;
                    found := true;
                    path[num_in_path] := j;
                    current_node := F[j];
                  end;
                if ((j in A) and (F[j] = current_node) and
                    (T[j] = next_node[current_node])) then
                  begin
                    num_in_path := num_in_path + 1;
                    found := true;
                    path[num_in_path] := j;
                    current_node := T[j];
                  end;
              end; { while not found }
          until current_node = l;
        end; { procedure find_path }

{*************************************************************************}

    begin
      Ahat := [];                                    { 0. Initialize }
      if cost[es] < 0 then
        begin
          Nhat := [ T[es] ];
          delta[T[es]] := x[es];
        end { if cost[es] < 0 }
      else  { cost[es] >= 0 }
        begin
          Nhat := [ F[es] ];
          delta[F[es]] := u[es] - x[es];
        end; { else }

      repeat
        psi1 := [];                  { 1. Determine Candidates for Tree }
        psi2 := [];
        for j := 1 to NumSlackArcs do
          begin
```

```
      if ((j<>es) and (cost[j] >= 0) and (x[j] < u[j]) and
          (T[j] in Nhat) and (not(F[j] in Nhat))) then
        psi1 := psi1 + [ j ];
      if ((j<>es) and (cost[j] <= 0) and (x[j] > 0) and
          (F[j] in Nhat) and (not(T[j] in Nhat))) then
        psi2 := psi2 + [ j ];
    end; { for j := 1 to NumSlackArcs }
  if ((psi1 + psi2) = [1) then
    begin
      no_cycle_flag := true;
      exit;
    end { if ((psi1 + psi2) = [1) }
  else
    no_cycle_flag := false;

  j := 0;                                  { 2. Append New Arc to Tree }
  found := false;
  repeat
    j := j + 1;
    if (j in psi1) then
      begin
        if (delta[T[j]] < (u[j] - x[j])) then
          delta[F[j]] := delta[T[j]]
        else
          delta[F[j]] := (u[j] - x[j]);
        found := true;
      end;
    if (j in psi2) then
      begin
        if (delta[F[j]] < x[j]) then
          delta[T[j]] := delta[F[j]]
        else
          delta[T[j]] := x[j];
        found := true;
      end;
  until found;
  Nhat := Nhat + [ T[j],F[j] ];
  Ahat := Ahat + [ j ];
until (F[es] in Nhat) and (T[es] in Nhat);

if (cost[es] < 0) then                          { 3. Breakthrough }
  begin
    find_path(T[es],F[es],Nhat,Ahat,tree_path,num_path_arcs);
    current_node := T[es];
    for j := 1 to num_path_arcs do
      begin
        if (T[tree_path[j]] = current_node) then
          begin
            x[tree_path[j]] := x[tree_path[j]] + delta[F[es]];
            current_node := F[tree_path[j]];
          end
        else { F[tree_path[j]] = current_node }
          begin
            x[tree_path[j]] := x[tree_path[j]] - delta[F[es]];
            current_node := T[tree_path[j]];
          end;
      end; { for j := 1 to num_path_arcs }
    x[es] := x[es] - delta[F[es]];
```

```
      end
    else  { cost[es] >= 0 }
      begin
        current_node := F[es];
        find_path(F[es],T[es],Nhat,Ahat,tree_path,num_path_arcs);
        for j := 1 to num_path_arcs do
          begin
            if (T[tree_path[j]] = current_node) then
              begin
                x[tree_path[j]] := x[tree_path[j]] + delta[F[es]];
                current_node := F[tree_path[j]];
              end
            else { F[tree_path[j]] = current_node }
              begin
                x[tree_path[j]] := x[tree_path[j]] - delta[F[es]];
                current_node := T[tree_path[j]];
              end;
          end; { for j := 1 to num_path_arcs }
        x[es] := x[es] + delta[T[es]];
      end;

  end; { procedure primal }

{*****************************************************************************}

  procedure dual(es : integer; var need_arc_flag : boolean);
  {
      This procedure implements the dual phase of the out of
      kilter algorithm on the tree developed in the primal
      phase and sets the flag need_arc_flag.  If arc es is in
      kilter at the end of the dual phase, a new out of kilter
      arc must be found, so need_arc_flag is set to true.  If
      arc es is still out of kilter at the end of the dual phase,
      then the primal phase must be executed again, and the
      need_arc_flag is set to false.
  }

  var
    i,j : integer;

  begin
  { Start with T = (Nhat,Ahat) }                    { 0. Initialization }
  { developed in primal phase  }

    psi1 := [];                          { 1. Determine Arcs Incident on T }
    psi2 := [];
    for j := 1 to NumSlackArcs do
      begin
        if (cost[j] < 0) and (not(F[j] in Nhat)) and
           (T[j] in Nhat) then
          psi1 := psi1 + [ j ];
        if (cost[j] > 0) and (F[j] in Nhat) and
           (not(T[j] in Nhat)) then
          psi2 := psi2 + [ j ];
      end; { for j := 1 to NumSlackArcs }

    theta := inf;              { 2. Determine Maximum Permissable Change }
    for j := 1 to NumSlackArcs do
```

```
          if (j in (psi1 + psi2)) and (Abs(cost[j]) < theta) then
            theta := Abs(cost[j]);

        for i := 1 to NumSlackNodes do                    { 3. Reduce Duals }
          if ( i in Nhat ) then
            pi[i] := pi[i] - theta;

        for j := 1 to NumSlackArcs do                       { Update Costs }
          cost[j] := pi[F[j]] - pi[T[j]] - c[j];

        need_arc_flag := true;                            { Set need_arc_flag }
        if (cost[es] < 0) and (x[es] > 0) then
          need_arc_flag := false;
        if (cost[es] > 0) and (x[es] < u[es]) then
          need_arc_flag := false;

      end; { procedure dual }

{*****************************************************************}

begin
  NumSlackNodes := NumNodes  + 1;
  NumSlackArcs := NumArcs + NumNodes;
  Initial_Solution;              { Find an initial set of feasible flows }
  need_arc := true;          { Do not currently know an out of kilter arc }
  repeat
    if need_arc then                        { Find an out of kilter arc s }
      begin
        ArcSearch(in_kilter,s);       { If no out of kilter arcs,
                                        in_kilter = true }
      end;
    if not in_kilter then
      begin
        primal(s,no_cycle);         { Execute the primal phase with arc s }
        need_arc := true;
        if (no_cycle) then
          begin
            dual(s,need_arc);      { If primal phase finds no cycles,
                                     execute dual phase }
          end;
      end; { if not in_kilter }
  until in_kilter;
end; { procedure MinCostFlow }

{*****************************************************************}

procedure Phase1;
{
     This procedure performs the Phase I portion of the problem, i.e.
     determination of the checker requirements.

     The checker requirements are returned in the global vector k and
     are output to the file Servers.out.
}

var
  OutputFile : text;
```

```
begin
  SetDate(FirstDay,LastDay,TotalStages);  { Asks for period to schedule }
  GetArrivals(NN,A);        { Read arrival data into A, * stages into NN }

  if (NN <> TotalStages) then   { if NN <> TotalStages, something wrong }
    begin
      ClearScreen;
      GotoXY(15,12);
      write('Number of stages in Arrivals.Dat does not match number');
      GotoXY(15,13);
      write('of stages from first to last day of scheduling period.');
      GotoXY(15,15);
      write('        Strike return to continue.');
      GotoXY(25,17);
      write('NN = ',NN);
      GotoXY(25,18);
      write('TotalStages = ',TotalStages);
      readln;
      Exit;
    end;

  InitPointers(NN);                                  { Initialize pointers }
  FillDPTable;   { Calculates f & d for the entire DP state-stage table }
  ForwardPass;           { Finds optimal path thru DP state-stage table }
  if TotalHours > MaxHours then
    begin
      ExceedHours := true;
      Deallocate;
    end
  else
    ExceedHours := false;
  rewrite(OutputFile,'Servers.out');                  { Open output file }
  case FirstDay of                          { Write FirstDay to output file }
    Monday : writeln(OutputFile,'Monday');
    Tuesday : writeln(OutputFile,'Tuesday');
    Wednesday : writeln(OutputFile,'Wednesday');
    Thursday : writeln(OutputFile,'Thursday');
    Friday : writeln(OutputFile,'Friday');
    Saturday : writeln(OutputFile,'Saturday');
    Sunday : writeln(OutputFile,'Sunday');
  end; { case FirstDay }
  for n := 1 to NN do       { Write optimal * checkers to 'Servers.out' }
    writeln(OutputFile,k[n]);
  Close(OutputFile);                                 { Close output file }
end; { procedure Phase1 }

{*****************************************************************************}

procedure Phase11;
{
     This procedure reads the first day of the scheduling period
     and the ideal checker requirements (output from Phase 1)
     from the file 'Servers.out'.  It then determines the optimal
     number of checkers to schedule for each shift.  These optimal
     shifts are output to the file 'Shifts.out'.
}

var
```

```
  InputFile : text;                    { File variable for 'Servers.out' }
  OutputFile : text;                   { File variable for 'Shifts.out'  }
  DayText : string[10];
  NumHours : integer;          { * of hours store is open for current day }
  NumShifts : integer;          { * possible daily shifts for current day }
  NumVars : integer;         { * possible daily shifts + * deviation vars }
  Num8 : integer;              { * possible 8 hour shifts for current day }
  Num7 : integer;              { * possible 7 hour shifts for current day }
  Num6 : integer;              { * possible 6 hour shifts for current day }
  Num5 : integer;              { * possible 5 hour shifts for current day }
  Num4 : integer;              { * possible 4 hour shifts for current day }
  shift_F : ArcArray;               { "From" functions for arcs (shifts) }
  shift_T : ArcArray;                 { "To" functions for arcs (shifts) }
  shift : ArcArray;                             { * servers in each shift }
  b : NodeArray;                      { change in hourly requirements }
  obj : ArcArray;              { cost coefficients in objective function }
  upper : ArcArray;                    { upper bounds on shift variables }
  shift_start : integer;              { start time for the current shift }
  shift_end : integer;                  { end time for the current shift }

begin
  reset(InputFile,'Servers.out');                          { Open Input File }
  readln(InputFile,DayText);       { Read first day of scheduling period }
  if DayText = 'Monday' then
    FirstDay := Monday;
  if DayText = 'Tuesday' then
    FirstDay := Tuesday;
  if DayText = 'Wednesday' then
    FirstDay := Wednesday;
  if DayText = 'Thursday' then
    FirstDay := Thursday;
  if DayText = 'Friday' then
    FirstDay := Friday;
  if DayText = 'Saturday' then
    FirstDay := Saturday;
  if DayText = 'Sunday' then
    FirstDay := Sunday;

 { Read checker requirements into k and total * stages into NN }
  n := 0;
  while not SeekEof(InputFile) do
    begin
      n := n + 1;
      if not SeekEoln(InputFile) then
        read(InputFile,k[n])
      else
        begin
          readln;
          read(InputFile,k[n])
        end;
    end;
  Close(InputFile);
  NN := n;

  rewrite(OutputFile,'Shifts.out');
  if ExceedHours then
    begin
      write(OutputFile,'*****************************************');
```

```
      writeln(OutputFile,'*****************************************');
      write(OutputFile,'WARNING     WARNING     WARNING     ');
      writeln(OutputFile,'WARNING     WARNING     WARNING     WARNING');
      writeln(OutputFile);
      writeln(OutputFile);
      write(OutputFile,'     Ideal checker requirements for each hour ');
      writeln(OutputFile,'exceed the maximum total hours.');
      writeln(OutputFile);
      write(OutputFile,'     Suggest you reduce target customer waiting ');
      writeln(OutputFile,'time and run program again.');
      writeln(OutputFile);
      writeln(OutputFile);
      write(OutputFile,'WARNING     WARNING     WARNING     ');
      writeln(OutputFile,'WARNING     WARNING     WARNING     WARNING');
      writeln(OutputFile);
      write(OutputFile,'*****************************************');
      writeln(OutputFile,'*****************************************');
      writeln(OutputFile);
      writeln(OutputFile);
    end;

  CurrentDay := FirstDay;
  n := 1;

  repeat

    NumHours := (time[CurrentDay].close - time[CurrentDay].open) div 100;
    if NumHours <> 0 then
      begin
        NumHours := NumHours + 1; { Add 1 hour for additional,
                                    redundant node }

        for i := 1 to NumHours do
          begin
            if i = 1 then
              b[i] := k[n]
            else
              if i = NumHours then
                b[i] := -k[n]
              else
                b[i] := k[n] - k[n-1];
            if i < NumHours then
              n := n + 1;
          end;

        Num8 := NumHours - 8;
        Num7 := NumHours - 7;
        Num6 := NumHours - 6;
        Num5 := NumHours - 5;
        Num4 := NumHours - 4;
        NumVars := Num8 + Num7 + Num6 + Num5 + Num4 +
                   (2 * (NumHours - 1));

        for j := 1 to Num8 do
          begin
            shift_F[j] := j;
            shift_T[j] := j + 8;
            obj[j] := 0;
```

```
      upper[j] := MAX_CHECKERS;
    end;
  for j := (Num8 + 1) to (Num8 + Num7) do
    begin
      shift_F[j] := j - Num8;
      shift_T[j] := j - Num8 + 7;
      obj[j] := 0;
      upper[j] := MAX_CHECKERS;
    end;
  for j := (Num8 + Num7 + 1) to (Num8 + Num7 + Num6) do
    begin
      shift_F[j] := j - (Num8 + Num7);
      shift_T[j] := j - (Num8 + Num7) + 6;
      obj[j] := 0;
      upper[j] := MAX_CHECKERS;
    end;
  for j := (Num8 + Num7 + Num6 + 1) to
           (Num8 + Num7 + Num6 + Num5) do
    begin
      shift_F[j] := j - (Num8 + Num7 + Num6);
      shift_T[j] := j - (Num8 + Num7 + Num6) + 5;
      obj[j] := 0;
      upper[j] := MAX_CHECKERS;
    end;
  for j := (Num8 + Num7 + Num6 + Num5 + 1) to
           (Num8 + Num7 + Num6 + Num5 + Num4) do
    begin
      shift_F[j] := j - (Num8 + Num7 + Num6 + Num5);
      shift_T[j] := j - (Num8 + Num7 + Num6 + Num5) + 4;
      obj[j] := 0;
      upper[j] := MAX_CHECKERS;
    end;

{ "From" & "To" functions for deviation variables }

  j := (Num8 + Num7 + Num6 + Num5 + Num4 + 1);
  for i := 1 to (NumHours-1) do
    begin
      shift_F[j] := i;                                    { di- }
      shift_T[j] := i + 1;
      obj[j] := 1;
      upper[j] := MAX_CHECKERS;
      shift_F[j+1] := i + 1;                              { di+ }
      shift_T[j+1] := i;
      obj[j+1] := 1;
      upper[j+1] := MAX_CHECKERS;
      j := j + 2;
    end;

  MinCostFlow(NumHours,NumVars,obj,shift,shift_F,shift_T,b,upper);

{ Write shifts to output file }
  writeln(OutputFile);
  case CurrentDay of
    Monday : writeln(OutputFile,'Monday');
    Tuesday : writeln(OutputFile,'Tuesday');
    Wednesday : writeln(OutputFile,'Wednesday');
    Thursday : writeln(OutputFile,'Thursday');
```

```
  Friday : writeln(OutputFile,'Friday');
  Saturday : writeln(OutputFile,'Saturday');
  Sunday : writeln(OutputFile,'Sunday');
end; { case CurrentDay of }

case CurrentDay of
  Monday : writeln('Monday');
  Tuesday : writeln('Tuesday');
  Wednesday : writeln('Wednesday');
  Thursday : writeln('Thursday');
  Friday : writeln('Friday');
  Saturday : writeln('Saturday');
  Sunday : writeln('Sunday');
end; { case CurrentDay of }

writeln(OutputFile,'     8 Hour Shifts:');
for j := 1 to Num8 do
  begin
    shift_start := time[CurrentDay].open + ((j - 1) * 100);
    shift_end := shift_start + (800);
    write(OutputFile,'          ');
    writeln(OutputFile,shift_start:4,'-',shift_end:4,
            '     ',shift[j]:2);
  end;

writeln(OutputFile,'     7 Hour Shifts:');
for j := (Num8 + 1) to (Num8 + Num7) do
  begin
    shift_start := time[CurrentDay].open +
                   ((j - Num8 - 1) * 100);
    shift_end := shift_start + (700);
    write(OutputFile,'          ');
    writeln(OutputFile,shift_start:4,'-',shift_end:4,
            '     ',shift[j]:2);
  end;

writeln(OutputFile,'     6 Hour Shifts:');
for j := (Num8 + Num7 + 1) to (Num8 + Num7 + Num6) do
  begin
    shift_start := time[CurrentDay].open +
                   ((j - Num8 - Num7 - 1) * 100);
    shift_end := shift_start + (600);
    write(OutputFile,'          ');
    writeln(OutputFile,shift_start:4,'-',shift_end:4,
            '     ',shift[j]:2);
  end;

writeln(OutputFile,'     5 Hour Shifts:');
for j := (Num8 + Num7 + Num6 + 1) to
         (Num8 + Num7 + Num6 + Num5) do
  begin
    shift_start := time[CurrentDay].open +
                   ((j - Num8 - Num7 - Num6 - 1) * 100);
    shift_end := shift_start + (500);
    write(OutputFile,'          ');
    writeln(OutputFile,shift_start:4,'-',shift_end:4,
            '     ',shift[j]:2);
  end;
```

```
        writeln(OutputFile,'     4 Hour Shifts:');
        for j := (Num8 + Num7 + Num6 + Num5 + 1) to
                 (Num8 + Num7 + Num6 + Num5 + Num4) do
          begin
            shift_start := time[CurrentDay].open +
                           ((j - Num8 - Num7 - Num6 - Num5 - 1) * 100);
            shift_end := shift_start + (400);
            write(OutputFile,'          ');
            writeln(OutputFile,shift_start:4,'-',shift_end:4,
                    '     ',shift[j]:2);
          end;

      end; { if NumHours <> 0 }

    if CurrentDay = Sunday then
      CurrentDay := Monday
    else
      CurrentDay := succ(CurrentDay);

  until (n >= NN);
  Close(OutputFile);
  writeln('Number of stages = ',n);
end; { procedure Phase II }

{*****************************************************************************}

procedure Solve;

begin
  Phase1;
  Phase11;
end; { procedure Solve }

{*****************************************************************************}

procedure SetHours;
{
    This procedure is used to modify the store's hours of operation.
}

var
  correct, BadResponse : boolean;
  day : DayOfWeek;
  response : char;

begin
  correct := false;
  while not correct do
    begin
      ClearScreen;                                  { ClrScr in IBM Turbo }
      GotoXY(1,7);
      writeln('                          Current hours of operation are:');
      writeln;
      writeln;
      writeln('                            Day        Open    Close');
      writeln;
      for day := Monday to Sunday do
```

```
        begin
          case day of
            Monday    : write('                              Monday  ');
            Tuesday   : write('                              Tuesday ');
            Wednesday : write('                             Wednesday');
            Thursday  : write('                             Thursday ');
            Friday    : write('                              Friday  ');
            Saturday  : write('                             Saturday ');
            Sunday    : write('                              Sunday  ')
          end; { case }
          if time[day].open = time[day].close then
            writeln('       Closed')
          else
            writeln(time[day].open:8,time[day].close:8);
        end; { day = Mon to Sun }
      writeln;
      writeln;
      BadResponse := true;
      while BadResponse do
        begin
          write('                          Is this correct (Y/N)?');
          readln(response);
          if (response = 'y') or (response = 'Y') then
            exit
          else
            if (response = 'n') or (response = 'N') then
              BadResponse := false
            else
              writeln('Please use: Y for yes, N for no');
        end; { BadResponse }
      ClearScreen;
      writeln;
      writeln;
      writeln('                  Please use 24-hour times for all entries.');
      writeln('                  Times must be rounded to nearest hour.');
      writeln;
      writeln('                  For days when store is closed, enter 0000');
      writeln('                  for opening time and 0000 for closing
time.');
      writeln;
      for day := Monday to Sunday do
        begin
          case day of
            Monday : writeln('                                  Monday');
            Tuesday : writeln('                                  Tuesday');
            Wednesday : writeln('                                  Wednesday');
            Thursday : writeln('                                  Thursday');
            Friday : writeln('                                  Friday');
            Saturday : writeln('                                  Saturday');
            Sunday : writeln('                                  Sunday');
          end;
          write('                                Open (xxxx):');
          readln(time[day].open);
          write('                               Close (xxxx):');
          readln(time[day].close);
        end;
    end; { not correct }
end; { procedure SetHours }
```

```
{*************************************************************************}

procedure SetMaxHours;
{
     This procedure prompts the user for the maximum allowable
     checker hours and stores it in the global variable MaxHours.
}
begin
  ClearScreen;
  GotoXY(15,12);
  write('Enter maximum allowable total checker hours:');
  readln(MaxHours);
end; { procedure SetMaxHours }

{*************************************************************************}

procedure SetTarget;
{
     This procedure prompts the user for the desired customer
     waiting time and stores it in the global variable target.
}
begin
  ClearScreen;
  GotoXY(20,12);
  write('Enter desired customer waiting time:');
  readln(target);
end;

{*************************************************************************}

begin

  InitHours;                          { Initialize the default store hours }
  InitTarget;     { Initialize the default desired customer waiting time }
  InitMaxHours;     { Initialize the default maximum total checker hours }

  done := false;
  repeat
    Menu(choice);                                          { Put up menu }
    case choice of
      1 : Phase1;
      2 : Phase11;
      3 : Solve;
      4 : SetHours;
      5 : SetTarget;
      6 : SetMaxHours;
      7 : done := true;
    end; { case }
  until done;
  Exit;

end.
```

Appendix B: Customer Arrival Data

Date	Day	Time	A(n)
27-May-87	Wednesday	9	240
27-May-87	Wednesday	10	312
27-May-87	Wednesday	11	347
27-May-87	Wednesday	12	329
27-May-87	Wednesday	13	326
27-May-87	Wednesday	14	323
27-May-87	Wednesday	15	319
27-May-87	Wednesday	16	384
27-May-87	Wednesday	17	337
27-May-87	Wednesday	18	131
28-May-87	Thursday	9	158
28-May-87	Thursday	10	263
28-May-87	Thursday	11	311
28-May-87	Thursday	12	270
28-May-87	Thursday	13	300
28-May-87	Thursday	14	292
28-May-87	Thursday	15	309
28-May-87	Thursday	16	369
28-May-87	Thursday	17	337
28-May-87	Thursday	18	275
28-May-87	Thursday	19	201
28-May-87	Thursday	20	57
29-May-87	Friday	9	62
29-May-87	Friday	10	153
29-May-87	Friday	11	170
29-May-87	Friday	12	227
29-May-87	Friday	13	258
29-May-87	Friday	14	291
29-May-87	Friday	15	316
29-May-87	Friday	16	322
29-May-87	Friday	17	278
29-May-87	Friday	18	114
30-May-87	Saturday	8	166
30-May-87	Saturday	9	290
30-May-87	Saturday	10	345
30-May-87	Saturday	11	354
30-May-87	Saturday	12	400
30-May-87	Saturday	13	350
30-May-87	Saturday	14	363

30-May-87	Saturday	15	332
30-May-87	Saturday	16	278
30-May-87	Saturday	17	114
31-May-87	Sunday	9	274
31-May-87	Sunday	10	395
31-May-87	Sunday	11	334
31-May-87	Sunday	12	307
31-May-87	Sunday	13	257
31-May-87	Sunday	14	318
31-May-87	Sunday	15	156
31-May-87	Sunday	16	3
2-Jun-87	Tuesday	9	174
2-Jun-87	Tuesday	10	281
2-Jun-87	Tuesday	11	332
2-Jun-87	Tuesday	12	323
2-Jun-87	Tuesday	13	339
2-Jun-87	Tuesday	14	329
2-Jun-87	Tuesday	15	357
2-Jun-87	Tuesday	16	394
2-Jun-87	Tuesday	17	317
2-Jun-87	Tuesday	18	209
2-Jun-87	Tuesday	19	26
3-Jun-87	Wednesday	9	118
3-Jun-87	Wednesday	10	171
3-Jun-87	Wednesday	11	222
3-Jun-87	Wednesday	12	195
3-Jun-87	Wednesday	13	195
3-Jun-87	Wednesday	14	259
3-Jun-87	Wednesday	15	305
3-Jun-87	Wednesday	16	306
3-Jun-87	Wednesday	17	320
3-Jun-87	Wednesday	18	77
4-Jun-87	Thursday	9	144
4-Jun-87	Thursday	10	300
4-Jun-87	Thursday	11	277
4-Jun-87	Thursday	12	308
4-Jun-87	Thursday	13	230
4-Jun-87	Thursday	14	260
4-Jun-87	Thursday	15	314
4-Jun-87	Thursday	16	383
4-Jun-87	Thursday	17	330
4-Jun-87	Thursday	18	219
4-Jun-87	Thursday	19	184

4-Jun-87	Thursday	20	52
5-Jun-87	Friday	9	157
5-Jun-87	Friday	10	263
5-Jun-87	Friday	11	317
5-Jun-87	Friday	12	286
5-Jun-87	Friday	13	287
5-Jun-87	Friday	14	304
5-Jun-87	Friday	15	319
5-Jun-87	Friday	16	275
5-Jun-87	Friday	17	286
5-Jun-87	Friday	18	96
6-Jun-87	Saturday	8	120
6-Jun-87	Saturday	9	235
6-Jun-87	Saturday	10	281
6-Jun-87	Saturday	11	345
6-Jun-87	Saturday	12	340
6-Jun-87	Saturday	13	334
6-Jun-87	Saturday	14	338
6-Jun-87	Saturday	15	306
6-Jun-87	Saturday	16	315
6-Jun-87	Saturday	17	129
7-Jun-87	Sunday	9	1
7-Jun-87	Sunday	10	244
7-Jun-87	Sunday	11	295
7-Jun-87	Sunday	12	280
7-Jun-87	Sunday	13	312
7-Jun-87	Sunday	14	353
7-Jun-87	Sunday	15	333
7-Jun-87	Sunday	16	124
9-Jun-87	Tuesday	9	197
9-Jun-87	Tuesday	10	295
9-Jun-87	Tuesday	11	303
9-Jun-87	Tuesday	12	304
9-Jun-87	Tuesday	13	316
9-Jun-87	Tuesday	14	278
9-Jun-87	Tuesday	15	306
9-Jun-87	Tuesday	16	352
9-Jun-87	Tuesday	17	309
9-Jun-87	Tuesday	18	186
9-Jun-87	Tuesday	19	53
10-Jun-87	Wednesday	9	119
10-Jun-87	Wednesday	10	234
10-Jun-87	Wednesday	11	277

10-Jun-87	Wednesday	12	249
10-Jun-87	Wednesday	13	238
10-Jun-87	Wednesday	14	275
10-Jun-87	Wednesday	15	271
10-Jun-87	Wednesday	16	259
10-Jun-87	Wednesday	17	246
10-Jun-87	Wednesday	18	0
11-Jun-87	Thursday	9	73
11-Jun-87	Thursday	10	188
11-Jun-87	Thursday	11	242
11-Jun-87	Thursday	12	184
11-Jun-87	Thursday	13	207
11-Jun-87	Thursday	14	284
11-Jun-87	Thursday	15	298
11-Jun-87	Thursday	16	302
11-Jun-87	Thursday	17	316
11-Jun-87	Thursday	18	213
11-Jun-87	Thursday	19	193
11-Jun-87	Thursday	20	58
12-Jun-87	Friday	9	173
12-Jun-87	Friday	10	235
12-Jun-87	Friday	11	292
12-Jun-87	Friday	12	276
12-Jun-87	Friday	13	255
12-Jun-87	Friday	14	317
12-Jun-87	Friday	15	307
12-Jun-87	Friday	16	301
12-Jun-87	Friday	17	310
12-Jun-87	Friday	18	97
13-Jun-87	Saturday	8	75
13-Jun-87	Saturday	9	189
13-Jun-87	Saturday	10	313
13-Jun-87	Saturday	11	328
13-Jun-87	Saturday	12	363
13-Jun-87	Saturday	13	361
13-Jun-87	Saturday	14	327
13-Jun-87	Saturday	15	366
13-Jun-87	Saturday	16	355
13-Jun-87	Saturday	17	145
14-Jun-87	Sunday	9	68
14-Jun-87	Sunday	10	236
14-Jun-87	Sunday	11	299
14-Jun-87	Sunday	12	339

14-Jun-87	Sunday	13	326
14-Jun-87	Sunday	14	336
14-Jun-87	Sunday	15	343
14-Jun-87	Sunday	16	159
16-Jun-87	Tuesday	9	200
16-Jun-87	Tuesday	10	322
16-Jun-87	Tuesday	11	300
16-Jun-87	Tuesday	12	327
16-Jun-87	Tuesday	13	279
16-Jun-87	Tuesday	14	302
16-Jun-87	Tuesday	15	371
16-Jun-87	Tuesday	16	334
16-Jun-87	Tuesday	17	313
16-Jun-87	Tuesday	18	199
16-Jun-87	Tuesday	19	59

Appendix C: SLAM Code and Output

Case 1: 6 Jun 87

```
GEN,CAPT FREY,D 6 JUN 87,11/16/88,1,Y,N,Y/Y,N,Y/S,72;
LIMITS,1,4,500;
NETWORK
      RESOURCE/CHECKERS(9),1;
      CREATE,XX(1),,1;
      ACTIVITY,,TNOW.LE.600,OK;
      ACTIVITY,,TNOW.GT.600,KIL;
KIL   TERM;
OK    ASSIGN,ATRIB(2)=5.156;
      ASSIGN,ATRIB(3)=ATRIB(1)+ATRIB(2);
      ASSIGN,ATRIB(4)=NNQ(1);
      AWAIT(1),CHECKERS/1;
      ACTIVITY/1,ATRIB(2);   SERVICE TIME
      FREE,CHECKERS/1;
      EVENT,1;
      COLCT,INT(1),TIME IN SYSTEM;
      COLCT,INT(3),TIME IN QUEUE;
      TERM;
;*************************************
      CREATE,60,60,,11;
      EVENT,2;
      ACTIVITY,,TNOW.GE.60 .AND. TNOW.LT.120,P2;
      ACTIVITY,,TNOW.GE.120 .AND. TNOW.LT.180,P3;
      ACTIVITY,,TNOW.GE.180 .AND. TNOW.LT.240,P4;
      ACTIVITY,,TNOW.GE.240 .AND. TNOW.LT.300,P5;
      ACTIVITY,,TNOW.GE.300 .AND. TNOW.LT.360,P6;
      ACTIVITY,,TNOW.GE.360 .AND. TNOW.LT.420,P7;
      ACTIVITY,,TNOW.GE.420 .AND. TNOW.LT.480,P8;
      ACTIVITY,,TNOW.GE.480 .AND. TNOW.LT.540,P9;
      ACTIVITY,,TNOW.GE.540 .AND. TNOW.LT.600,P10;
      ACTIVITY,,TNOW.GE.600 .AND. TNOW.LT.660,P11;
      ACTIVITY,,TNOW.GE.660,P12;
P2    ASSIGN,XX(1)=60.0/235.0;
      ALTER,CHECKERS/+11;
      TERMINATE;
P3    ASSIGN,XX(1)=60.0/281.0;
      ALTER,CHECKERS/+3;
      TERMINATE;
P4    ASSIGN,XX(1)=60.0/345.0;
      ALTER,CHECKERS/+7;
      TERMINATE;
P5    ASSIGN,XX(1)=60.0/340.0;
      ALTER,CHECKERS/-1;
      TERMINATE;
P6    ASSIGN,XX(1)=60.0/334.0;
      TERMINATE;
P7    ASSIGN,XX(1)=60.0/338.0;
      ALTER,CHECKERS/+1;
      TERMINATE;
P8    ASSIGN,XX(1)=60.0/306.0;
      ALTER,CHECKERS/-5;
      TERMINATE;
```

```
53  P9      ASSIGN,XX(1)=60.0/315.0;
54          ALTER,CHECKERS/+3;
55          TERMINATE;
56  P10     ASSIGN,XX(1)=60.0/129.0;
57          ALTER,CHECKERS/-15;
58          TERMINATE;
59  P11     ALTER,CHECKERS/-12;
60          TERMINATE;
61  P12     ASSIGN,XX(1)=100000;
62          TERMINATE;
63          ENDNETWORK;
64  INTLC,XX(1)=0.5,XX(2)=0,XX(3)=0,XX(4)=0,XX(5)=0,XX(6)=0;
65  INIT,0.0,800.0,N;
66  FIN;
```

S L A M I I S U M M A R Y R E P O R T

SIMULATION PROJECT D 6 JUN 87 BY CAPT FREY

DATE 11/16/1988 RUN NUMBER 1 OF 1

CURRENT TIME 0.8000E+03
STATISTICAL ARRAYS CLEARED AT TIME 0.0000E+00

STATISTICS FOR VARIABLES BASED ON OBSERVATION

	MEAN VALUE	STANDARD DEVIATION	COEFF. OF VARIATION	MINIMUM VALUE	MAXIMUM VALUE	NO.OF OBS
TIME IN SYSTEM	0.787E+01	0.111E+01	0.141E+00	0.516E+01	0.124E+02	2743
TIME IN QUEUE	0.271E+01	0.111E+01	0.409E+00	0.000E+00	0.722E+01	2743

FILE STATISTICS

FILE NUMBER	LABEL/TYPE	AVERAGE LENGTH	STANDARD DEVIATION	MAXIMUM LENGTH	CURRENT LENGTH	AVERAGE WAIT TIME
1	AWAIT	9.304	7.508	28	0	2.714
2	CALENDAR	19.504	12.193	33	1	0.931

REGULAR ACTIVITY STATISTICS

ACTIVITY INDEX/LABEL	AVERAGE UTILIZATION	STANDARD DEVIATION	MAXIMUM UTIL	CURRENT UTIL	ENTITY COUNT
1 SERVICE TIME	17.6787	11.9310	30	0	2743

RESOURCE STATISTICS

RESOURCE NUMBER	RESOURCE LABEL	CURRENT CAPACITY	AVERAGE UTIL	STANDARD DEVIATION	MAXIMUM UTIL	CURRENT UTIL
1	CHECKERS	1	17.68	11.931	30	0

RESOURCE NUMBER	RESOURCE LABEL	CURRENT AVAILABLE	AVERAGE AVAILABLE	MINIMUM AVAILABLE	MAXIMUM AVAILABLE
1	CHECKERS	1	0.2713	-15	9

Case 2: 6 Jun 87

S L A M I I S U M M A R Y R E P O R T

SIMULATION PROJECT Con Lamda 1 BY CAPT FREY

DATE 11/16/1988 RUN NUMBER 50 OF 50

CURRENT TIME 0.8000E+03
STATISTICAL ARRAYS CLEARED AT TIME 0.0000E+00

STATISTICS FOR VARIABLES BASED ON OBSERVATION

	MEAN VALUE	STANDARD DEVIATION	COEFF. OF VARIATION	MINIMUM VALUE	MAXIMUM VALUE	NO.OF OBS
TIME IN SYSTEM	0.117E+02	0.102E+02	0.874E+00	0.155E+01	0.230E+03	****
TIME IN QUEUE	0.655E+01	0.100E+02	0.153E+01	0.000E+00	0.227E+03	****

FILE STATISTICS

FILE NUMBER	LABEL/TYPE	AVERAGE LENGTH	STANDARD DEVIATION	MAXIMUM LENGTH	CURRENT LENGTH	AVERAGE WAIT TIME
1	AWAIT	12.801	12.971	52	0	3.764
2	CALENDAR	19.155	11.770	33	1	0.889

REGULAR ACTIVITY STATISTICS

ACTIVITY INDEX/LABEL	AVERAGE UTILIZATION	STANDARD DEVIATION	MAXIMUM UTIL	CURRENT UTIL	ENTITY COUNT
1 SERVICE TIME	17.6153	11.6713	30	0	2721

RESOURCE STATISTICS

RESOURCE NUMBER	RESOURCE LABEL	CURRENT CAPACITY	AVERAGE UTIL	STANDARD DEVIATION	MAXIMUM UTIL	CURRENT UTIL
1	CHECKERS	1	17.62	11.671	30	0

RESOURCE NUMBER	RESOURCE LABEL	CURRENT AVAILABLE	AVERAGE AVAILABLE	MINIMUM AVAILABLE	MAXIMUM AVAILABLE
1	CHECKERS	1	1.4318	-15	18

Case 3: 6 Jun 87

S L A M I I S U M M A R Y R E P O R T

SIMULATION PROJECT Con Lamda 1 BY CAPT FREY

DATE 11/16/1988 RUN NUMBER 50 OF 50

CURRENT TIME 0.8000E+03
STATISTICAL ARRAYS CLEARED AT TIME 0.0000E+00

STATISTICS FOR VARIABLES BASED ON OBSERVATION

	MEAN VALUE	STANDARD DEVIATION	COEFF. OF VARIATION	MINIMUM VALUE	MAXIMUM VALUE	NO.OF OBS
TIME IN SYSTEM	0.131E+02	0.123E+02	0.940E+00	0.159E+01	0.260E+03	****
TIME IN QUEUE	0.798E+01	0.122E+02	0.153E+01	0.000E+00	0.256E+03	****

FILE STATISTICS

FILE NUMBER	LABEL/TYPE	AVERAGE LENGTH	STANDARD DEVIATION	MAXIMUM LENGTH	CURRENT LENGTH	AVERAGE WAIT TIME
1	AWAIT	20.763	21.055	85	0	6.014
2	CALENDAR	19.519	12.029	33	1	0.894

REGULAR ACTIVITY STATISTICS

ACTIVITY INDEX/LABEL	AVERAGE UTILIZATION	STANDARD DEVIATION	MAXIMUM UTIL	CURRENT UTIL	ENTITY COUNT
1 SERVICE TIME	17.6009	11.6214	30	0	2762

RESOURCE STATISTICS

RESOURCE NUMBER	RESOURCE LABEL	CURRENT CAPACITY	AVERAGE UTIL	STANDARD DEVIATION	MAXIMUM UTIL	CURRENT UTIL
1	CHECKERS	1	17.60	11.621	30	0

RESOURCE NUMBER	RESOURCE LABEL	CURRENT AVAILABLE	AVERAGE AVAILABLE	MINIMUM AVAILABLE	MAXIMUM AVAILABLE
1	CHECKERS	1	1.2355	-15	18

Case 4: 6 Jun 87

S L A M I I S U M M A R Y R E P O R T

SIMULATION PROJECT Con Lamda 1 BY CAPT FREY

DATE 11/16/1988 RUN NUMBER 50 OF 50

CURRENT TIME 0.8000E+03
STATISTICAL ARRAYS CLEARED AT TIME 0.0000E+00

STATISTICS FOR VARIABLES BASED ON OBSERVATION

	MEAN VALUE	STANDARD DEVIATION	COEFF. OF VARIATION	MINIMUM VALUE	MAXIMUM VALUE	NO.OF OBS
TIME IN SYSTEM	0.172E+02	0.178E+02	0.103E+01	0.155E+01	0.285E+03	*****
TIME IN QUEUE	0.121E+02	0.177E+02	0.147E+01	0.000E+00	0.282E+03	*****

FILE STATISTICS

FILE NUMBER	LABEL/TYPE	AVERAGE LENGTH	STANDARD DEVIATION	MAXIMUM LENGTH	CURRENT LENGTH	AVERAGE WAIT TIME
1	AWAIT	42.949	25.762	92	7	12.554
2	CALENDAR	19.729	11.764	33	2	0.914

REGULAR ACTIVITY STATISTICS

ACTIVITY INDEX/LABEL	AVERAGE UTILIZATION	STANDARD DEVIATION	MAXIMUM UTIL	CURRENT UTIL	ENTITY COUNT
1 SERVICE TIME	17.5882	11.5355	30	1	2729

RESOURCE STATISTICS

RESOURCE NUMBER	RESOURCE LABEL	CURRENT CAPACITY	AVERAGE UTIL	STANDARD DEVIATION	MAXIMUM UTIL	CURRENT UTIL
1	CHECKERS	1	17.59	11.535	30	1

RESOURCE NUMBER	RESOURCE LABEL	CURRENT AVAILABLE	AVERAGE AVAILABLE	MINIMUM AVAILABLE	MAXIMUM AVAILABLE
1	CHECKERS	0	0.9292	-15	21

Case 1: 11 Jun 87

S L A M I I S U M M A R Y R E P O R T

SIMULATION PROJECT D 11 JUN 87 BY CAPT FREY

DATE 11/16/1988 RUN NUMBER 1 OF 1

CURRENT TIME 0.8000E+03
STATISTICAL ARRAYS CLEARED AT TIME 0.0000E+00

STATISTICS FOR VARIABLES BASED ON OBSERVATION

	MEAN VALUE	STANDARD DEVIATION	COEFF. OF VARIATION	MINIMUM VALUE	MAXIMUM VALUE	NO.OF OBS
TIME IN SYSTEM	0.876E+01	0.175E+01	0.199E+00	0.516E+01	0.167E+02	2558
TIME IN QUEUE	0.361E+01	0.175E+01	0.484E+00	0.000E+00	0.115E+02	2558

FILE STATISTICS

FILE NUMBER	LABEL/TYPE	AVERAGE LENGTH	STANDARD DEVIATION	MAXIMUM LENGTH	CURRENT LENGTH	AVERAGE WAIT TIME
1	AWAIT	11.535	7.857	33	0	3.608
2	CALENDAR	18.386	8.817	30	1	0.955

REGULAR ACTIVITY STATISTICS

ACTIVITY INDEX/LABEL	AVERAGE UTILIZATION	STANDARD DEVIATION	MAXIMUM UTIL	CURRENT UTIL	ENTITY COUNT
1 SERVICE TIME	16.4863	8.6246	27	0	2558

RESOURCE STATISTICS

RESOURCE NUMBER	RESOURCE LABEL	CURRENT CAPACITY	AVERAGE UTIL	STANDARD DEVIATION	MAXIMUM UTIL	CURRENT UTIL
1	CHECKERS	1	16.49	8.625	27	0

RESOURCE NUMBER	RESOURCE LABEL	CURRENT AVAILABLE	AVERAGE AVAILABLE	MINIMUM AVAILABLE	MAXIMUM AVAILABLE
1	CHECKERS	1	0.1887	-11	5

Case 2: 11 Jun 87

S L A M I I S U M M A R Y R E P O R T

SIMULATION PROJECT Con Lamda 2 BY CAPT FREY

DATE 11/16/1988 RUN NUMBER 50 OF 50

CURRENT TIME 0.8000E+03
STATISTICAL ARRAYS CLEARED AT TIME 0.0000E+00

STATISTICS FOR VARIABLES BASED ON OBSERVATION

	MEAN VALUE	STANDARD DEVIATION	COEFF. OF VARIATION	MINIMUM VALUE	MAXIMUM VALUE	NO.OF OBS
TIME IN SYSTEM	0.121E+02	0.786E+01	0.649E+00	0.159E+01	0.154E+03	****
TIME IN QUEUE	0.698E+01	0.759E+01	0.109E+01	0.000E+00	0.150E+03	****

FILE STATISTICS

FILE NUMBER	LABEL/TYPE	AVERAGE LENGTH	STANDARD DEVIATION	MAXIMUM LENGTH	CURRENT LENGTH	AVERAGE WAIT TIME
1	AWAIT	5.856	5.943	27	0	1.858
2	CALENDAR	18.051	8.452	30	1	0.951

REGULAR ACTIVITY STATISTICS

ACTIVITY INDEX/LABEL	AVERAGE UTILIZATION	STANDARD DEVIATION	MAXIMUM UTIL	CURRENT UTIL	ENTITY COUNT
1 SERVICE TIME	16.3874	8.4792	27	0	2522

RESOURCE STATISTICS

RESOURCE NUMBER	RESOURCE LABEL	CURRENT CAPACITY	AVERAGE UTIL	STANDARD DEVIATION	MAXIMUM UTIL	CURRENT UTIL
1	CHECKERS	1	16.39	8.479	27	0

RESOURCE NUMBER	RESOURCE LABEL	CURRENT AVAILABLE	AVERAGE AVAILABLE	MINIMUM AVAILABLE	MAXIMUM AVAILABLE
1	CHECKERS	1	0.4144	-11	16

Case 3: 11 Jun 87

S L A M I I S U M M A R Y R E P O R T

SIMULATION PROJECT Con Lamda 2 BY CAPT FREY

DATE 11/16/1988 RUN NUMBER 50 OF 50

CURRENT TIME 0.8000E+03
STATISTICAL ARRAYS CLEARED AT TIME 0.0000E+00

STATISTICS FOR VARIABLES BASED ON OBSERVATION

	MEAN VALUE	STANDARD DEVIATION	COEFF. OF VARIATION	MINIMUM VALUE	MAXIMUM VALUE	NO.OF OBS
TIME IN SYSTEM	0.139E+02	0.101E+02	0.726E+00	0.156E+01	0.162E+03	****
TIME IN QUEUE	0.872E+01	0.986E+01	0.113E+01	0.000E+00	0.157E+03	****

FILE STATISTICS

FILE NUMBER	LABEL/TYPE	AVERAGE LENGTH	STANDARD DEVIATION	MAXIMUM LENGTH	CURRENT LENGTH	AVERAGE WAIT TIME
1	AWAIT	16.298	14.204	63	0	5.101
2	CALENDAR	18.463	8.482	30	1	0.960

REGULAR ACTIVITY STATISTICS

ACTIVITY INDEX/LABEL	AVERAGE UTILIZATION	STANDARD DEVIATION	MAXIMUM UTIL	CURRENT UTIL	ENTITY COUNT
1 SERVICE TIME	16.4270	8.4610	27	0	2556

RESOURCE STATISTICS

RESOURCE NUMBER	RESOURCE LABEL	CURRENT CAPACITY	AVERAGE UTIL	STANDARD DEVIATION	MAXIMUM UTIL	CURRENT UTIL
1	CHECKERS	1	16.43	8.461	27	0

RESOURCE NUMBER	RESOURCE LABEL	CURRENT AVAILABLE	AVERAGE AVAILABLE	MINIMUM AVAILABLE	MAXIMUM AVAILABLE
1	CHECKERS	1	0.3357	-11	15

Case 4: 11 Jun 87

S L A M I I S U M M A R Y R E P O R T

SIMULATION PROJECT Con Lamda 2 BY CAPT FREY

DATE 11/16/1988 RUN NUMBER 50 OF 50

CURRENT TIME 0.8000E+03
STATISTICAL ARRAYS CLEARED AT TIME 0.0000E+00

STATISTICS FOR VARIABLES BASED ON OBSERVATION

	MEAN VALUE	STANDARD DEVIATION	COEFF. OF VARIATION	MINIMUM VALUE	MAXIMUM VALUE	NO.OF OBS
TIME IN SYSTEM	0.163E+02	0.137E+02	0.838E+00	0.156E+01	0.170E+03	****
TIME IN QUEUE	0.111E+02	0.135E+02	0.121E+01	0.000E+00	0.166E+03	****

FILE STATISTICS

FILE NUMBER	LABEL/TYPE	AVERAGE LENGTH	STANDARD DEVIATION	MAXIMUM LENGTH	CURRENT LENGTH	AVERAGE WAIT TIME
1	AWAIT	10.989	12.032	54	0	3.579
2	CALENDAR	17.826	8.451	30	1	0.964

REGULAR ACTIVITY STATISTICS

ACTIVITY INDEX/LABEL	AVERAGE UTILIZATION	STANDARD DEVIATION	MAXIMUM UTIL	CURRENT UTIL	ENTITY COUNT
1 SERVICE TIME	16.2872	8.4402	27	0	2456

RESOURCE STATISTICS

RESOURCE NUMBER	RESOURCE LABEL	CURRENT CAPACITY	AVERAGE UTIL	STANDARD DEVIATION	MAXIMUM UTIL	CURRENT UTIL
1	CHECKERS	1	16.29	8.440	27	0

RESOURCE NUMBER	RESOURCE LABEL	CURRENT AVAILABLE	AVERAGE AVAILABLE	MINIMUM AVAILABLE	MAXIMUM AVAILABLE
1	CHECKERS	1	0.4695	-11	17

Bibliography

Baker, Kenneth R. "Workforce Allocation in Cyclical Scheduling Problems: A Survey," Operational Research Quarterly, 27: 155-167 (1976).

Bartholdi, John J. "A Guaranteed-Accuracy Round-off Algorithm for Cyclic Scheduling and Set Covering," Operations Research, 29: 501-510 (May-June 1981).

Bellman, Richard. Dynamic Programming. Princeton, NJ: Princeton University Press, 1957.

Denardo, Eric V. Dynamic Programming: Models and Applications. Englewood Cliffs, NJ: Prentice-Hall, Inc., 1982.

Fisher, Marshall L. "The Lagrangian Relaxation Method for Solving Integer Programming Problems," Management Science, 27: 1-18 (January 1981).

Fulkerson, D. R. "An Out-of-Kilter Method for Minimal-Cost Flow Problems," Journal of the Society of Industrial and Applied Mathematics, 9: 18-27 (1961).

Hillier, Frederick S. and Gerald J. Lieberman. Introduction to Operations Research (Fourth Edition). Oakland, CA: Holden-Day, Inc., 1986.

Kleinrock, Leonard. Queueing Systems, Volume II: Computer Applications. New York: John Wiley and Sons, Inc., 1976.

Koelling, C. Patrick and James E. Bailey. "A Multiple Criteria Decision Aid for Personnel Scheduling," IIE Transactions, 16: 299-307 (December 1984).

Kwan, Stephen K. , Mark M. Davis, and Allen G. Greenwood. "A Simulation Model for Determining Variable Worker Requirements in a Service Operation with Time-Dependent Customer Demand," Queueing Systems, 3: 265-276 (1988).

Larson, Robert E. and John L. Casti. Principles of Dynamic Programming, Part II: Advanced Theory and Applications. New York: Marcel Dekker, Inc., 1982.

Magazine, M. J. "Optimal Control of Multi-channel Service Systems," Naval Research Logistics Quartly, 18: 177-183 (1971).

Moulder, Capt Roger D. Development of a General Purpose Commissary Store Simulation Model. MS thesis, AFIT/GOR/ENS/87D-11. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December, 1987.

Polk, Lt Col Stan. Telephone interview. HQ AFCOMS, Kelly AFB TX, 26 May 1988.

Segal, M. "The Operator-Scheduling Problem: A Network-Flow Approach," Operations Research, 22: 808-823 (July-August 1974).

Veinott, Arthur F. and Harvey M. Wagner. "Optimal Capacity Scheduling," Operations Research, 10: 518-532 (July-August 1962).

VITA

Captain Thomas J. Frey [redacted]

Pennsylvania, in 1980 [redacted] attended Pennsylvania State University, from which he received the degree of Bachelor of Science in Electrical Engineering in May 1984. After graduation, he attended Officer Training School, and he received a commission in the USAF in August 1984. He was an electronic countermeasures signal analyst at the Foreign Technology Division at Wright-Patterson AFB from September 1984 until May 1987, at which time he entered the School of Engineering, Air Force Institute of Technology.

[redacted]

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release distribution unlimited.
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) AFIT/GOR/ENS/88D-7	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION School of Engineering 6b. OFFICE SYMBOL (If applicable) AFIT/ENS	7a. NAME OF MONITORING ORGANIZATION
6c. ADDRESS (City, State, and ZIP Code) Air Force Institute of Technology Wright-Patterson AFB OH 45433-6583	7b. ADDRESS (City, State, and ZIP Code)
8a. NAME OF FUNDING/SPONSORING ORGANIZATION AF Commissary Service 8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (City, State, and ZIP Code) Kelly AFB TX 78241-5000	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO / WORK UNIT ACCESSION NO.
11. TITLE (Include Security Classification) See Box 19	
12. PERSONAL AUTHOR(S) Thomas J. Frey, Captain, USAF	
13a. TYPE OF REPORT MS Thesis 13b. TIME COVERED FROM _____ TO _____ 14. DATE OF REPORT (Year, Month, Day) 1988 December 15. PAGE COUNT 111	
16. SUPPLEMENTARY NOTATION	

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	
12	04		Queueing Theory, Scheduling, Dynamic Programming, Network Flows

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

Title: OPTIMAL SERVER SCHEDULING TO MAINTAIN CONSTANT CUSTOMER WAITING TIMES

Thesis Advisor: Joseph Litko, Major, USAF
Assistant Professor of Operations Research

Approved for public release IAW AFR 190-1
10 Jan 89

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED
22a. NAME OF RESPONSIBLE INDIVIDUAL Joseph Litko, Major, USAF	22b. TELEPHONE (Include Area Code) (513)255-3362 22c. OFFICE SYMBOL AFIT/ENS

DD Form 1473, JUN 86 *Previous editions are obsolete.*

SECURITY CLASSIFICATION OF THIS PAGE
UNCLASSIFIED

The purpose of this research was to develop an analytical model that would optimally schedule commissary checkers so that the expected customer waiting-time would remain relatively constant throughout the scheduling period. A two-phase model was developed to solve the problem. The first phase of the model used dynamic programming to find the optimal number of checkers required throughout each day to meet the desired customer waiting-time goal. Since checkers cannot be scheduled to work arbitrarily short tours of duty, a second phase was needed in the model to find the optimal number of checkers to assign to allowable shifts in order to meet the optimal requirements determined in phase one.

A simulation was developed to validate the checker scheduling model. It was found that the scheduling model produced acceptable results until the last few periods of the day. Additional servers needed to be added heuristically near the end of each day to obtain the desired customer waiting times.

Several extensions of this work are possible. First, an improved approximation for customer line lengths could be used at the end of each day. Use of such an approximation could eliminate the need for heuristic rules in scheduling servers during the last few periods of each day. Second, the scheduling algorithm that was developed did not account for checker lunch breaks. Accounting for lunch breaks complicates the problem, but two different approaches were suggested for a solution allowing for checker lunch breaks. Finally, a third phase could be added to the model that would allow assignment of actual workers to the optimal shifts determined in the second phase.